SOLUTION OF REGULAR, SPARSE TRIANGULAR LINEAR SYSTEMS ON VECTOR AND DISTRIBUTED-MEMORY MULTIPROCESSORS

E. Barszcz[*] R. Fatoohi[†] V. Venkatakrishnan[†] S. Weeratunga[†]

NAS Applied Research Branch
NASA Ames Research Center, Moffett Field, CA 94035

Report RNR-93-007, April 1993

Abstract

This paper presents the implementations and results of a model problem, the Symmetric Successive Over-Relaxation (SSOR) simulated application benchmark from the NAS Parallel Benchmark suite for three different parallel processors. SSOR is an iterative implicit method that partitions the left hand side matrix into a lower triangular matrix and an upper triangular matrix. The machines used are an eight processor Cray Y-MP, a 32k processor Thinking Machines Corp. CM-2 and a 128 processor Intel iPSC/860. The primary difficulty in implementing SSOR on a parallel machine lies in finding enough parallelism within the triangular solves to keep a large number of processors active. A data mapping useful for distributed memory architectures is presented. The results show that the eight processor Cray Y-MP has the best performance among the three machines.

[*]The author is an employee of NASA.

[†]The author is an employee of Computer Sciences Corp. This work was funded through NASA contract NAS 2-12961

1 INTRODUCTION

In this paper, implementations and results of a model problem, the Symmetric Successive Over-Relaxation (SSOR) simulated application benchmark from the NAS Parallel Benchmark suite [2] are presented for three different parallel processors. For a detailed description of the mathematical definition of the SSOR simulated application refer to [2]. The machines used are an eight processor Cray Y-MP, a 32k processor Thinking Machines Corp. CM-2 and a 128 processor Intel iPSC/860. All machines are located at the Numerical Aerodynamic Simulation (NAS) systems division at NASA Ames Research Center.

The SSOR simulated application benchmark models one of the solution methods used in computational fluid dynamics. SSOR is an iterative implicit method that partitions the left hand side matrix into a lower triangular matrix and an upper triangular matrix. A steady state solution is found by forming the right hand side vector, forming and solving the lower triangular system of equations, forming and solving the upper triangular system of equation and then updating the solution. These four steps are iterated until convergence. The primary difficulty in implementing SSOR on a parallel machine lies in finding enough parallelism within the triangular solves to keep a large number of processors active.

In the remainder of the paper, the model problem is presented and solution algorithms for the triangular solves are reviewed. The Cray Y-MP implementation and results are also presented. In addition, a data mapping useful for distributed memory architectures is presented. Finally, the Thinking Machines Corp. CM-2 and Intel iPSC/860 implementations and results are given.

2 MODEL PROBLEM

The model problem used for this investigation is the Symmetric Successive Over-Relaxation simulated application benchmark in the NAS Parallel Benchmark suite. For a detailed description of the mathematical definition refer to [2]. Here we provide only a brief description of the essential features of the problem for completeness.

We consider the numerical solution of the following system of five second-order, nonlinear partial differential equations (PDE's):

$$\begin{aligned}\frac{\partial \mathbf{U}}{\partial \tau} &= \frac{\partial \mathbf{E}(\mathbf{U})}{\partial \xi} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial \eta} + \frac{\partial \mathbf{G}(\mathbf{U})}{\partial \zeta} \\ &\quad + \frac{\partial \mathbf{T}(\mathbf{U}, \mathbf{U}_\xi)}{\partial \xi} + \frac{\partial \mathbf{V}(\mathbf{U}, \mathbf{U}_\eta)}{\partial \eta} + \frac{\partial \mathbf{W}(\mathbf{U}, \mathbf{U}_\zeta)}{\partial \zeta} \\ &\quad + \mathbf{H}(\mathbf{U}, \mathbf{U}_\xi, \mathbf{U}_\eta, \mathbf{U}_\zeta), \;\; (\tau, \xi, \eta, \zeta) \in D_\tau \times D\end{aligned} \tag{1}$$

with the uncoupled Dirichlet boundary conditions:

$$\mathbf{U} = \mathbf{U}^*(\xi, \eta, \zeta), \;\; (\tau, \xi, \eta, \zeta) \in D_\tau \times \partial D \tag{2}$$

and initial conditions:

$$\mathbf{U} = \mathbf{U}^0(\xi, \eta, \zeta), \;\; (\xi, \eta, \zeta) \in D \, for \; \tau = 0, \tag{3}$$

where $D \subseteq \Re^3$ is a bounded domain, ∂D is its boundary and $D_\tau = \{0 \leq \tau \leq T\}$.

Here, $\mathbf{E}$, $\mathbf{F}$, $\mathbf{G}$, $\mathbf{T}$, $\mathbf{V}$, $\mathbf{W}$ and $\mathbf{H}$ are prescribed vector funtions of $\mathbf{U}$, $\mathbf{U}_\xi$, $\mathbf{U}_\eta$ and $\mathbf{U}_\zeta$ with five components each. The vector funtions $\mathbf{U}^*$ and $\mathbf{U}^0$ are given and their exact form can be found in [2].

In this study, we seek a steady-state solution to (2) of the form:

$$\mathbf{U} = [u^{(1)}\ u^{(2)}\ u^{(3)}\ u^{(4)}\ u^{(5)}]^T = \mathbf{U}^*(\xi,\eta,\zeta), \ \ (\xi,\eta,\zeta) \in D.$$

To this end, the vector forcing function $\mathbf{H}$ is chosen such that the system of PDE's along with its boundary conditions, satisfies the prescribed exact solution $\mathbf{U}^*$. The bounded spatial domain D is specified to be the interior of the unit cube $[(0,1) \times (0,1) \times (0,1)]$ and its boundary ∂D, is the surface of the unit cube.

Starting from the initial values $\mathbf{U}^0$, we seek some discrete approximation $\mathbf{U}_h^* \in \Re^3$ to the steady-state solution $\mathbf{U}^*$. This is achieved through the numerical solution of the nonlinear system of PDE's using a time marching scheme and a spatial discretization procedure based on second-order accurate, centered finite-difference approximations.

The independent temporal variable τ is discretized with $\Delta\tau$ as the uniform increment and the discrete approximation of $\mathbf{U}$ on D_τ is denoted by:

$$\mathbf{U}(\tau) \approx \mathbf{U}^\tau(n\Delta\tau) = \mathbf{U}^n.$$

Implicit time differencing is achieved through a single-step, two-level, first-order accurate Euler scheme given by:

$$\Delta\mathbf{U}^n = \Delta\tau\frac{\partial(\Delta\mathbf{U}^n)}{\partial\tau} + \Delta\tau\frac{\partial\mathbf{U}^n}{\partial\tau} + O[\Delta\tau^2]. \tag{4}$$

where,

$$\Delta\mathbf{U}^n = \mathbf{U}^{n+1} - \mathbf{U}^n.$$

After (4) is linearized using a local Taylor series expansion in time about $\mathbf{U}^n$ (followed by some simplification) the following linear equation for $\Delta\mathbf{U}^n$ is obtained:

$$\{\mathbf{I} - \Delta\tau[\frac{\partial(\mathbf{A})^n}{\partial\xi} + \frac{\partial^2(\mathbf{N})^n}{\partial\xi^2} + \frac{\partial(\mathbf{B})^n}{\partial\eta} + \frac{\partial^2(\mathbf{Q})^n}{\partial\eta^2} + \frac{\partial(\mathbf{C})^n}{\partial\zeta} + \frac{\partial^2(\mathbf{S})^n}{\partial\zeta^2}]\}\Delta\mathbf{U}^n =$$
$$\Delta\tau[\frac{\partial(\mathbf{E}+\mathbf{T})^n}{\partial\xi} + \frac{\partial(\mathbf{F}+\mathbf{V})^n}{\partial\eta} + \frac{\partial(\mathbf{G}+\mathbf{W})^n}{\partial\zeta}] + \Delta\tau\mathbf{H}^*, \tag{5}$$

where $\mathbf{A}(\mathbf{U})$, $\mathbf{B}(\mathbf{U})$, $\mathbf{C}(\mathbf{U})$, $\mathbf{N}(\mathbf{U})$, $\mathbf{Q}(\mathbf{U})$ and $\mathbf{S}(\mathbf{U})$ are the (5×5) Jacobian matrices $(\partial\mathbf{E}/\partial\mathbf{U})$, $(\partial\mathbf{F}/\partial\mathbf{U})$, $(\partial\mathbf{G}/\partial\mathbf{U})$, $(\partial\mathbf{T}/\partial\mathbf{U})$, $(\partial\mathbf{V}/\partial\mathbf{U})$ and $(\partial\mathbf{W}/\partial\mathbf{U})$. Exact forms of these matrices can be found in [2]. Then the solution at the next time step is given by:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \Delta\mathbf{U}^n.$$

The independent spatial variables (ξ,η,ζ) are discretized by covering $\bar{D}$ (the closure of D), with a mesh of uniform increments (h_ξ, h_η, h_ζ) in each of the coordinate directions, Figure 1. The mesh points in the region are identified by the index-triple (i,j,k), where the indices $i \in [1,N_\xi]$, $j \in [1,N_\eta]$ and $k \in [1,N_\zeta]$ correspond to the

Figure 1: Computational Domain.

discretization of ξ, η and ζ coordinates respectively. Then N_ξ, N_η and N_ζ are the number of mesh points in ξ, η and ζ directions respectively and:

$$h_\xi = 1/(N_\xi - 1);\ h_\eta = 1/(N_\eta - 1);\ \ h_\zeta = 1/(N_\zeta - 1).$$

Also, the discrete approximation of $\mathbf{U}$ in $(\bar{D} \times D_\tau)$ is denoted by:

$$\mathbf{U}(\tau, \xi, \eta, \zeta) \cong U_h^\tau[n\Delta\tau, (i-1)h_\xi, (j-1)h_\eta, (k-1)h_\zeta] = U_{i,j,k}^n. \tag{6}$$

The spatial derivatives in (5) are approximated by the appropriate three-point, second-order accurate centered finite-difference quotients, based on the values of $\mathbf{U}_h^\tau$ at mesh points in $D_h \cup \partial D_h$. Details can be found in [2].

In addition, to suppress high-frequency modes and allow the numerical scheme to converge to a steady state, a linear fourth-difference dissipation term of the following form:

$$-\Delta\tau\, \varepsilon\, [h_\xi^4 D_\xi^4 \mathbf{U}^n + h_\eta^4 D_\eta^4 \mathbf{U}^n + h_\zeta^4 D_\zeta^4 \mathbf{U}^n],$$

is added to the right hand side of (5). Here, ε is a specified constant, and

$$h_\xi^4 D_\xi^4 \mathbf{U}^n = \nabla_\xi \Delta_\xi \nabla_\xi \Delta_\xi \mathbf{U}_{i,j,k}{}^n, \tag{7}$$

where Δ_ξ, ∇_ξ are the standard forward and backward difference operators associated with the ξ direction, respectively. Similar equations hold for the η and ζ directions.

At the first two interior mesh points belonging to either end of the computational domain, the standard five point difference stencil used for the fourth-difference dissipation term is replaced by one-sided or one-side biased stencils. These modifications maintain a non-positive definite dissipation matrix for the system of difference equations [2].

The one-sided and one-side biased dissipation stencils for interior grid points near the boundaries in the ξ direction are given by the following equations
at i=2:

$$h_\xi^4 D_\xi^4 \mathbf{U}^n = \mathbf{U}_{i+2,j,k}^n - 4\mathbf{U}_{i+1,j,k}^n + 5\mathbf{U}_{i,j,k}^n, \tag{8}$$

and at i=3:

$$h_\xi^4 D_\xi^4 \mathbf{U}^n = \mathbf{U}_{i+2,j,k}^n - 4\mathbf{U}_{i+1,j,k}^n + 6\mathbf{U}_{i,j,k}^n - 4\mathbf{U}_{i-1,j,k}^n, \tag{9}$$

at $i = N_\xi - 2$:

$$h_\xi^4 D_\xi^4 \mathbf{U}^n = -4\mathbf{U}_{i+1,j,k}^n + 6\mathbf{U}_{i,j,k}^n - 4\mathbf{U}_{i-1,j,k}^n + \mathbf{U}_{i-2,j,k}^n, \tag{10}$$

and at $i = N_\xi - 1$:

$$h_\xi^4 D_\xi^4 \mathbf{U}^n = 5\mathbf{U}_{i,j,k}^n - 4\mathbf{U}_{i-1,j,k}^n + \mathbf{U}_{i-2,j,k}^n. \tag{11}$$

Similar difference formulas are used in the η and ζ directions.

Then, adding the dissipation terms to the right hand side of (5) and replacing the derivative operators by the difference operators D and D^2 yields the linear system of equations for $\Delta\mathbf{U}^n$ given by:

$$\begin{aligned}\{\mathbf{I} - \Delta\tau[D_\xi(\mathbf{A})^n + D_\xi^2(\mathbf{N})^n + D_\eta(\mathbf{B})^n + D_\eta^2(\mathbf{Q})^n + D_\zeta(\mathbf{C})^n + D_\zeta^2(\mathbf{S})^n]\}\Delta\mathbf{U}^n = \\ \Delta\tau[D_\xi(\mathbf{E}+\mathbf{T})^n + D_\eta(\mathbf{F}+\mathbf{V})^n + D_\zeta(\mathbf{G}+\mathbf{W})^n] \\ - \Delta\tau\,\varepsilon\,[h_\xi^4 D_\xi^4 \mathbf{U}^n + h_\eta^4 D_\eta^4 \mathbf{U}^n + h_\zeta^4 D_\zeta^4 \mathbf{U}^n] + \Delta\tau\mathbf{H}^*,\end{aligned} \tag{12}$$

where $\mathbf{H}^*$ is the appropriately modified vector forcing function to account for the added fourth-difference dissipation. Also, D_ξ, and D_ξ^2 are the standard three-point central spatial difference operators, defined in D_h. The left hand side discretization involves a 7-point finite difference stencil, whereas the right hand side discretization involves a 13-point finite difference stencil, as shown in Figures 2 and 3, respectively. It should be noted that the structure of these stencils gets modified near the boundaries of the computational domain as a result of the modification to the fourth-difference dissipation terms.

2.1 STRUCTURE OF THE SYSTEM OF LINEAR EQUATIONS

When difference operators are replaced by their equivalent discrete forms, the following equation represents a system of discretized linear equations for $[\Delta\mathbf{U}^n]_{i,j,k}$, where $i \in [2, N_\xi - 1]$, $j \in [2, N_\eta - 1]$ and $k \in [2, N_\zeta - 1]$. Let this system of equations be denoted by:

$$\mathcal{K}^n \Delta\mathbf{U}^n = \mathbf{R}^n, \tag{13}$$

where $\Delta\mathbf{U}$ and $\mathbf{R}$ are each vectors of length $N = 5 \times (N_\xi - 2) \times (N_\eta - 2) \times (N_\zeta - 2)$, obtained by assembling the sub-vectors of length five associated with each internal mesh point of the grid. $\mathcal{K}$ is an $(N \times N)$ sparse matrix, whose nonzero elements are (5×5) sub-matrices. The maximum number of non-zeros per row of the matrix

Figure 2: 7-point finite-difference stencil for left hand side operator.

is 35 ($= 5 \times 7$), which is directly related to the 7-point difference stencil used for discretization of the left hand side of (13).

When mesh points are ordered in a systematic manner, the matrix $\mathcal{K}$ assumes a particular structure. In the present study, the mesh points are assumed to be in natural or lexicographic ordering: mesh points are numbered sequentially such that the index in ξ direction runs fastest, followed by the index in η direction and then the index in ζ direction. This results in a regular, sparse $(N \times N)$ block matrix with (5×5) sub-matrices as elements. The structure of $\mathcal{K}$ for a 3-D, $(4 \times 4 \times 4)$ regular mesh is shown in Figure 4. It is clear that $\mathcal{K}$ has a triply nested block tridiagonal form, and is structurally symmetric, although the matrix itself is non-symmetric.

The matrix $\mathcal{K}$ can be considered as having a sparse, block banded structure, with seven block-diagonals, each with (5×5) sub-matrices as their elements. This is shown in Figure 4. Let us denote the nodal variables grouped by sub-vectors of length five and (5×5) sub-matrix elements of $\mathcal{K}$, associated with a mesh point, by using its index triple (i, j, k) as the subscripts. Then the constituent equation associated with the mesh point (i, j, k) is given by:

$$\begin{aligned}\mathcal{A}_{i,j,k}\Delta\mathbf{U}_{i,j,k-1} + \mathcal{B}_{i,j,k}\Delta\mathbf{U}_{i,j-1,k} + \mathcal{C}_{i,j,k}\Delta\mathbf{U}_{i-1,j,k} + \mathcal{D}_{i,j,k}\Delta\mathbf{U}_{i,j,k} \\ +\mathcal{E}_{i,j,k}\Delta\mathbf{U}_{i+1,j,k} + \mathcal{F}_{i,j,k}\Delta\mathbf{U}_{i,j+1,k} + \mathcal{G}_{i,j,k}\Delta\mathbf{U}_{i,j,k+1} = \mathbf{R}_{i,j,k}\end{aligned} \tag{14}$$

where, $i \in [2, N_\xi - 1]$, $j \in [2, N_\eta - 1]$ and $k \in [2, N_\zeta - 1]$. Each of the coefficients

Figure 3: 13-point finite-difference stencil for the explicit part.

$\mathcal{A}(\mathbf{U}_{i,j,k-1}), \mathcal{B}(\mathbf{U}_{i,j-1,k}), \mathcal{C}(\mathbf{U}_{i-1,j,k}), \mathcal{D}(\mathbf{U}_{i,j,k}), \mathcal{E}(\mathbf{U}_{i+1,j,k}), \mathcal{F}(\mathbf{U}_{i,j+1,k})$ and $\mathcal{G}(\mathbf{U}_{i,j,k+1})$ is a (5×5) sub-matrix, whereas $\Delta\mathbf{U}_{i,j,k}$ and $\mathbf{R}_{i,j,k}$ are sub-vectors of length 5. In addition, the coefficients satisfy the following conditions near the boundaries of the computational domain:

$$\mathcal{A}_{i,j,k} = 0 \text{ for } k \leq 2; \;\; \mathcal{B}_{i,j,k} = 0 \text{ for } j \leq 2; \;\; \mathcal{C}_{i,j,k} = 0 \text{ for } i \leq 2;$$

$$\mathcal{E}_{i,j,k} = 0 \text{ for } i \geq (N_\xi - 1); \;\; \mathcal{F}_{i,j,k} = 0 \text{ for } j \geq (N_\eta - 1); \;\; \mathcal{G}_{i,j,k} = 0 \text{ for } k \geq (N_\zeta - 1).$$

3 SOLUTION OF SPARSE SYSTEM OF EQUATIONS

Since the direct solution of this system of linear system of order N requires a formidable matrix inversion effort, in terms of both the processing time and storage requirements, $\Delta\mathbf{U}^n$ is obtained through the use of an iterative method based on an approximation to the matrix $\mathcal{K}$, (13). In this study, we use the Symmetric Successive Over-Relaxation (SSOR) scheme with natural ordering. The matrix $\mathcal{K}$ can be written as the sum of the matrices $\mathcal{D}$, $\mathcal{Y}$ and $\mathcal{Z}$;

$$\mathcal{K}^n = \mathcal{D}^n + \mathcal{Y}^n + \mathcal{Z}^n$$

where,

$$\mathcal{D}^n \;\; = \;\; \text{main block-diagonal of } \mathcal{K}^n,$$

Figure 4: Schematic of matrix $\mathcal{K}$ under natural ordering of mesh points.

$$\begin{aligned}\mathcal{Y}^n &= \text{three sub-block diagonals of } \mathcal{K}^n,\\ \mathcal{Z}^n &= \text{three super-block diagonals of } \mathcal{K}^n.\end{aligned}$$

Therefore, $\mathcal{D}$ is a block-diagonal matrix, while $\mathcal{Y}$ and $\mathcal{Z}$ are strictly lower and upper triangular, respectively. Then the point-SSOR iterative scheme can be written as, [1];

$$\mathcal{X}^n \, \Delta\mathbf{U}^n = \mathbf{R}^n$$

where,

$$\begin{aligned}\mathcal{X}^n &= \omega(2-\omega)(\mathcal{D}^n + \omega\mathcal{Y}^n)(\mathcal{D}^n)^{-1}(\mathcal{D}^n + \omega\mathcal{Z}^n)\\ &= \omega(2-\omega)(\mathcal{D}^n + \omega\mathcal{Y}^n)[I + \omega(\mathcal{D}^n)^{-1}\mathcal{Z}^n]\end{aligned}$$

and $\omega \in (0,2)$ is the over-relaxation factor, a specified constant. The SSOR algorithm requires the solution of the following two sparse block lower and upper triangular linear systems:

$$[\mathcal{D}^n + \omega\mathcal{Y}^n]\Delta\mathbf{U}_1 = \mathbf{R}^n \tag{15}$$

$$[I + \omega(\mathcal{D}^n)^{-1}\mathcal{Z}^n]\Delta\mathbf{U}^n = \Delta\mathbf{U}_1 \tag{16}$$

so that,

$$\mathbf{U}^{n+1} = \mathbf{U}^n + [1/\omega(2-\omega)]\Delta\mathbf{U}^n.$$

The solution scheme consists of following computational tasks, per iteration:

a) Formation of the right hand side vector $\mathbf{R}$.

b) Generation of the sparse, lower and upper triangular systems and their inversion.

c) Solution update.

Efficient solution of the sparse triangular systems presents a challenging problem on vector and highly parallel computers due to the limited degree of concurrency associated with the underlying solution algorithms. This is in contrast to the high degree of easily exploitable concurrency available when forming $\mathbf{R}$, coefficient matrices of the triangular systems and the solution update. As a result, when mapping the computation onto distributed memory multiprocessors, one has to find a data partitioning scheme that does not unduly lower the efficiency or the concurrency associated with the computation of the vector $\mathbf{R}$, and at the same time allows the optimal concurrent solution of the triangular systems.

3.1 CONCURRENT ALGORITHMS FOR SPARSE TRIANGULAR LINEAR SYSTEM SOLUTION

In this section, we briefly review the various concurrent algorithms available for the solution of regular sparse, block lower and upper triangular systems. In the following discussion, we focus on the solution of sparse, lower triangular systems. Similar remarks apply to the sparse, upper triangular systems as well.

The constituent equation of the lower triangular system represented by (15) at a mesh point (i, j, k) is given by:

$$\omega\mathcal{A}_{i,j,k}\Delta\mathbf{U}_{i,j,k-1} + \omega\mathcal{B}_{i,j,k}\Delta\mathbf{U}_{i,j-1,k} + \omega\mathcal{C}_{i,j,k}\Delta\mathbf{U}_{i-1,j,k} + \mathcal{D}_{i,j,k}\Delta\mathbf{U}_{i,j,k} = \mathbf{R}_{i,j,k},$$

where, $i \in [2, N_\xi - 1], j \in [2, N_\eta - 1]$ and $k \in [2, N_\zeta - 1]$. This system can be solved by the following well-known recursive, sequential algorithm:

Algorithm 3.1 (Sequential Algorithm)

```
for $k = 2, \cdots, (N_\zeta - 1)$ do
    for $j = 2, \cdots, (N_\eta - 1)$ do
        for $i = 2, \cdots, (N_\xi - 1)$ do
            $\Delta\mathbf{U}_{i,j,k} = \mathcal{D}^{-1}_{i,j,k}[\mathbf{R}_{i,j,k} - \omega(\mathcal{A}_{i,j,k}\Delta\mathbf{U}_{i,j,k-1} + \mathcal{B}_{i,j,k}\Delta\mathbf{U}_{i,j-1,k} + \mathcal{C}_{i,j,k}\Delta\mathbf{U}_{i-1,j,k})]$
        enddo i
    enddo j
enddo k
```

This algorithm involves first-order linear recurrences (data dependencies) in all three indices and clearly lacks any concurrency. Also, since the recurrences occur in all three indices, re-ordering the indices does not alleviate the problem. Partial concurrency can be achieved by noting that the k-index recurrence refers to the $\Delta\mathbf{U}$-values on the previous (i, j)-plane. For a given value of the k index, the contributions to $\Delta\mathbf{U}_{i,j,k}$ from the previously computed index $(k - 1)$ are accumulated, resulting in $(N_\xi - 2)(N_\eta - 2)$ concurrent tasks. Similarly, for given values of indices k and j, one can proceed to accumulate the contributions to $\Delta\mathbf{U}_{i,j,k}$ from an already computed index $(j - 1)$. This results in $(N_\xi - 2)$ concurrent tasks. Finally, $\Delta\mathbf{U}_{i,j,k}$ is obtained by solving the remaining first-order linear recurrence in index i. This results in the following partially concurrent algorithm:

Figure 5: Schematic of Mesh Diagonals.

Based on this observation, concurrency in the execution of recursions over both i

and j indices can be achieved using the mesh-diagonal or the wavefront approach. In the wavefront approach, for a fixed value of k, unknowns are computed successively along the diagonals of the mesh on an $(i,j)-$plane, Figure 5. Also, concurrency for the recurrence in the k-index is achieved as before. For a fixed value of k, a wavefront is defined by the set:

$$W_m = \{(i,j) \mid i+j=m\}, \;\; m = 4,5,\cdots,(N_\xi + N_\eta - 2).$$

All the unknowns on a wavefront W_m can be computed concurrently, after all unknowns on wavefront W_{m-1} have been updated. This requires sweeping through all the mesh diagonals of a $(i,j)-$plane sequentially, while achieving concurrency within each wavefront. The number of concurrent tasks within a wavefront is variable and as a result, the degree of concurrency for this part of the algorithm ranges from a minimum of one to a maximum of $Min(N_\xi - 2, N_\eta - 2)$. This leads to the following concurrent algorithm:

Algorithm 3.3 (Wavefront Algorithm)

```
define sets $W_m = \{(i,j)|i+j=m\}, m = 4,5,\cdots,(N_\xi + N_\eta - 2)$
  for $k = 2,3,\cdots,(N_\zeta - 1)$ do
    for $j = 2,3,\cdots,(N_\eta - 1)$ do (concurrent loop)
      for $i = 2,3,\cdots,(N_\xi - 1)$ do (concurrent loop)
        $\Delta\mathbf{U}_{i,j,k} = \mathbf{R}_{i,j,k} - \omega\mathcal{A}_{i,j,k}\Delta\mathbf{U}_{i,j,k-1}$
      enddo i
    enddo j
    for $m = 4,5,\cdots,(N_\xi + N_\eta - 2)$ do
      for $(i,j) \in W_m$ do (concurrent loop)
        $\Delta\mathbf{U}_{i,j,k} = D^{-1}_{i,j,k}[\Delta\mathbf{U}_{i,j,k} - \omega(\mathcal{B}_{i,j,k}\Delta\mathbf{U}_{i,j-1,k} + \mathcal{C}_{i,j,k}\Delta\mathbf{U}_{i-1,j,k})]$
      enddo
    enddo m
  enddo k
```

The data dependency in (15) is such that, the solution for the unknown $\Delta\mathbf{U}_{i,j,k}$ requires only the knowledge of solutions $\Delta\mathbf{U}_{i,j,k-1}$, $\Delta\mathbf{U}_{i,j-1,k}$ and $\Delta\mathbf{U}_{i-1,j,k}$. Therefore, as soon as all the $\Delta\mathbf{U}$ values with $i+j+k=(m-1)$ are known for a given value of m, then all ΔU values with $i+j+k=m$ can be computed concurrently. This is the diagonal hyperplane method. A hyperplane H_m, for a fixed m, is defined by the set:

$$H_m = \{(i,j,k), |i+j+k=m\}, \;\; m = 6,7,\cdots,(N_\xi + N_\eta + N_\zeta - 3). \qquad (17)$$

All unknowns belonging to H_m can be computed concurrently, after all unknowns belonging to the previous hyperplane, H_{m-1} have been computed. As in the wavefront algorithm, this requires sequential sweeping through all the hyperplanes, while achieving concurrency within each hyperplane. Again, the number of concurrent tasks within a hyperplane is variable and consequently, the degree of concurrency for this algorithm begins with a minimum of one and increases to at most $Min\{(N_\xi - 2)(N_\eta - 2), (N_\eta - 2)(N_\zeta - 2), (N_\zeta - 2)(N_\xi - 2)\}$ and then reduces back to a minimum of one. The diagonal hyperplane algorithm can be written as follows:

Algorithm 3.4 (Hyperplane Algorithm)

```
define sets $H_m = \{(i,j,k) | i+j+k=m\}, m = 6, 7, \cdots, (N_\xi + N_\eta + N_\zeta - 3)$
    for $m = 6, 7, \cdots, (N_\xi + N_\eta + N_\zeta - 3)$ do
        for $(i,j,k) \in H_m$ do (concurrent loop)
            $\Delta\mathbf{U}_{i,j,k} = \mathcal{D}^{-1}_{i,j,k}[\mathbf{R}_{i,j,k} - \omega(\mathcal{A}_{i,j,k}\Delta\mathbf{U}_{i,j,k-1} + \mathcal{B}_{i,j-1,k}\Delta\mathbf{U}_{i,j-1,k} + \mathcal{C}_{i-1,j,k}\Delta\mathbf{U}_{i-1,j,k})]$
        enddo
    enddo m
```

It should be noted that in all of the above concurrent algorithms, we are only changing the order of computation by implicitly re-ordering the unknowns without changing the results. In essence, the round-off effects are the same as in the original sequential algorithm and the results are bitwise identical. However, the order in which the solution is computed differs from the standard lexicographical ordering given in Algorithm 3.1.

A higher degree of concurrency for the solution of sparse triangular systems can be achieved through different orderings of the unknowns. However, such ordering schemes are known to adversely affect the convergence rate of the underlying iterative scheme. The impact on the rate of convergence is difficult to study. Whether the gain in efficiency due to enhanced concurrency can outweigh the loss incurred by a reduced convergence rate is highly dependent on the nature of the problem being solved (the eigenspectrum of the matrix) and architectural features of the parallel computer system. In this study, we do not explore such algorithms.

4 DEFINITIONS

The following terms are used throughout the rest of the paper unless explicitly redefined.

Domain refers to the computational domain. For cubic domains, N is the length of the axes and the size is specified as $(N \times N \times N)$. For noncubic domains, N_ξ, N_η, and N_ζ are the lengths of the ξ, η, and ζ axes respectively and the size is specified as $(N_\xi \times N_\eta \times N_\zeta)$.

Related to parallel processing, P is the number of processors used, N_l is the local length of an axis that has been partitioned across P processors, $(\frac{N}{P})$, T is the execution time in seconds, T_1 is the execution time in seconds on a single processor, T_P is the execution time in seconds on P processors, S is the speedup defined as $\frac{T_1}{T_P}$, and e is the efficiency defined as $\frac{S}{P}$.

The processing rate (Mflop/s) is calculated by taking the best Cray single processor version and using the hardware performance monitor to measure the number of floating point operations and then dividing by the actual execution time in seconds times one million. This method is independent of machine and number of processors used: all processing rates use the best single processor Cray Y-MP floating point operations count irrespective of the number of operations actually performed.

Figure 6: Schematic of hyperplanes.

From a geometric point of view, the set of nodes (mesh points) belonging to each hyperplane lies on a plane askew to all three axes of the computational space Figure 6. During the solution of the lower triangular linear system, each concurrent computation occurs entirely within a hyperplane. However, a given hyperplane cannot be activated until all previous hyperplanes have completed their computation. This results in a sequential traversing through hyperplanes from one corner to the

Figure 7: Mesh Diagonals on a hyperplane.

diametrically opposite corner of the 3-D mesh. Each hyperplane is composed of a set of mesh diagonals and each of these mesh diagonals in turn consists of nodes whose k index is the same, see Figure 7. Assuming that all arrays are stored in the standard Fortran lexicographical ordering, the stride for accessing data on a given mesh diagonal is $(N_\xi - 1)$. However, when moving from the last node on the k-th mesh diagonal to the first node on the $(k+1)-$st mesh diagonal, a relatively large jump in the address space of the vector $\Delta\mathbf{U}$ is needed. In addition, the size of this jump varies with the value of k index.

The non-constant stride data access opens up at least two approaches for organizing vector-mode computations within a given hyperplane H_m. In the first case, vector operations can be confined to each of the mesh diagonals, ignoring the concurrency available across the mesh diagonals on a given hyperplane. This approach yields vector operations of variable and relatively short length, but with a constant stride of $(N_\xi - 1)$. The vector lengths encountered would range from a minimum of one to a maximum of $Min\{(N_\xi - 2), (N_\eta - 2)\}$. The number of vector start-ups required is equal to the number of mesh diagonals on a hyperplane. The relatively short vector lengths combined with the large number of start-ups required may limit the performance of this implementation on vector computers with a large start-up cost to asymptotic rate ratio [8]. However, this approach may prove attractive on vector processors that lack the hardware necessary to deal with vector memory accesses involving indirect addressing.

Because the data for all nodes on a given hyperplane H_m cannot be accessed by a constant vector stride, the second approach involves the use of indirect addressing to enable vector operations for all nodes on a hyperplane, simultaneously. This results in a single vector loop for each hyperplane, whose vector length changes as the hyperplanes are swept from one corner of the mesh to the other. The vector length ranges from a minimum of 1 to a maximum of $Min\{(N_\xi - 2)(N_\eta - 2), (N_\eta - 2)(N_\zeta - 2), (N_\zeta - 2)(N_\xi - 2)\}$ and then back again to 1.

The success of indirect addressing depends on the availability of hardware to handle indirect addressing efficiently. Its implementation requires the explicit use of index vectors to link nodes within a hyperplane to the index triple (i, j, k). All eligible nodes within a given hyperplane are sequentially ordered, beginning with those on the mesh diagonal corresponding to minimum k index. Along a given mesh diagonal, numbering begins at the end corresponding to minimum j-index. Based on this numbering scheme, a set of index vectors is generated, that links a node's position within the hyperplane it belongs to the index triple (i, j, k). This facilitates the implementation of the hyperplane algorithm with all the node variables stored using standard Fortran lexicographic ordering. The indirect addressing of vector $\mathbf{U}$ is needed when generating the elemental sub-matrices of the sparse triangular systems, while $\Delta\mathbf{U}$ is accessed during the solution for the nodal unknowns.

In both of the above approaches, the computation of the right hand side vector involves no indirect addressing and can be trivially vectorized with vector lengths of either $(N_\xi - 2)$ or $(N_\eta - 2)$ or $(N_\zeta - 2)$, all involving either unit or constant stride memory access.

If the nodal data is stored explicitly according to the sequential numbering scheme within a hyperplane H_m, indirect addressing can be avoided for data belonging to H_m. However, data belonging to nodes $(i, j, k-1)$, $(i, j-1, k)$ and $(i-1, j, k)$, all lying on H_{m-1}, cannot be accessed with a constant stride. This is because, in 3-D, node orderings on adjacent hyperplanes are not compatible. In addition, this storage scheme would require either indirect addressing during the computation of the right hand side vector or maintaining two copies of the nodal data, one in the hyperplane ordering and the other in natural ordering, with indirect addressing for movement of data between the two copies. These factors make this approach unattractive on the Cray Y-MP, which has excellent hardware support for accessing data using indirect addressing.

5.2 PARALLELIZATION ISSUES

In this section, issues pertaining to simultaneous vectorization and coarse-grain parallelization of sparse lower and upper triangular solvers on a shared memory multiprocessor are considered. The objective is optimum utilization of all processors without sacrificing vector performance.

Assuming a cubic domain, the average number of concurrent tasks available in the second part of the wavefront algorithm, Algorithm 3.3, is $O(N)$. This granularity is insufficient for optimum utilization of the vector pipelines on multiple processors or to amortize the overhead associated with creation and management of concurrent processes. Therefore the wavefront algorithm will not be considered any further in this section.

For the hyperplane algorithm, Algorithm 3.4, the average number of concurrent tasks within a hyperplane is $O(N^2)$. This makes the hyperplane algorithm an excellent choice for simultaneous vectorization and coarse-grained parallelization. Again, there are two approaches for implementation. As mentioned in Section 5.1, restricting the vectorization on a hyperplane H_m to mesh diagonals would leave enough concurrent tasks available for exploiting coarse grain parallelism across the mesh diagonals belonging to H_m. However, the task granularity of each mesh diagonal is not a constant. This would require that each mesh diagonal be assigned to the processors in a self-scheduled manner for effective load balancing. Since the number of mesh diagonals on hyperplanes close to the two corners of the 3-D mesh is small, some form of load imbalance is unavoidable. In addition, as mentioned in Section 5.1, vector performance for this approach is not optimal.

The second approach to implementing the hyperplane algorithm is based on indirect addressing. With indirect addressing, all nodes within a hyperplane are processed in a single do-loop. Concurrent tasks are formed by partitioning the do-loop into P or fewer equal-sized blocks and assigning each block to one of the P processors, which processes its block in vector-mode. This leads to more flexibility in partitioning and consequently to better load balancing. However, again near the two corners of the 3-D mesh, the number of nodes in a hyperplane is small and this leads to some load imbalance.

Generation of the coefficient matrices for the triangular systems can be dealt with in a similar manner. The only remaining steps in the algorithm, computation of the right hand side vector and the solution update, are trivially parallelized as well as vectorized.

5.2.1 Y-MP SYSTEM DESCRIPTION

The Cray Y-MP/8 has eight processors, 128 Mwords of main memory, and 6 nanoseconds clock cycle. The peak performance of the machine is 2.67 Gflop/s.

5.2.2 Y-MP IMPLEMENTATION DETAILS

The SSOR simulated application is multitasked on the Y-MP by autotasking the hyperplane algorithm, Algorithm 3.4. Autotasking, or automatic multitasking, is a technique where the compiling system attempts to detect and exploit parallelism in a Fortran program [4]. This process is automatic but not all types of parallelism are detected and programmer intervention is frequently required.

Autotasking works on do-loop boundaries. If there are no data dependencies in a nested do-loop, autotasking turns the nested loops to a vector inner loop and a parallel outer loop. With user direction, autotasking can also stripmine a single vectorized do-loop. Unlike multitasking independent nested loops, stripmining reduces the vector length of the inner loop and so affects vectorization performance, especially for short vectors.

The SSOR simulated application has two major parts: the right hand side (RHS) and the left hand side (LHS). The RHS is mainly written with triple nested loops where each loop has length N. This part is highly parallel with no data dependency in the three directions. The LHS part consists of the formation and solution of sparse lower and upper triangular matrices. The LHS has double nested loops where the inner loops are of lengths ranging between one and $\frac{3}{4}N^2$ with an average length of $\frac{1}{3}N^2$ and outer loops of length $3N - 2$.[1] In the LHS, only the inner loops have no data dependency.

The SSOR simulated application is multitasked by parallelizing the outer loops of the RHS and stripmining the inner loops of the LHS. This is similar to the approach taken by Fatoohi and Yoon [6] in multitasking the INS3D-LU code on the Cray Y-MP.

5.2.3 Y-MP RESULTS

Results are presented for several different domain sizes with varying numbers of processors. Table 1 contains the memory requirements in megabytes, measured execution time per time step in seconds, speedup, parallel efficiency, and the processing rate for four domains on P processors of the Cray Y-MP, where P is 1, 2, 4, or 8. The domains are all cubical with size $(N \times N \times N)$, where N is 32, 64, 102, or 128. All timings are measured in a dedicated environment. The required memory is measured by using the Unix command *size*, and verified by counting the number of the arrays in the code. Speedup, parallel efficiency and processing rate are computed as defined in Section 4.

The single processor results show that the SSOR simulated application has achieved reasonable performance even for the smallest domain where the inner loops in the RHS are of length 32. The SSOR simulated application achieved about 60% of the peak performance of single processor of the Cray Y-MP. This is because all of the inner loops are well vectorized, and have reasonable lengths.

Table 2 lists the measured times for the RHS, lower triangular solve, and upper triangular solve of the SSOR simulated application on the Cray Y-MP. These three parts account for approximately 99% of the execution time in each time step. Again, all timings are measured in a dedicated environment.

The parallel efficiency of the RHS and LHS (lower and upper triangular solves) are given in Table 3. These results show that the parallel efficiency of the algorithm depends primarily on the parallel efficiency of the LHS since the parallel efficiency of the RHS changes only slightly with an increase in the domain size. The RHS is highly parallel and the slight increase in parallel efficiency is due to the increase in task granularity for larger domains. For small domains, the LHS has many small loops which are both vectorized and multitasked. For example, the LHS of the $(32 \times 32 \times 32)$ domain has inner loops with a maximum length of 675 and an average length of 340. When stripmined across eight processors, the average vector length is 42. This explains the relative low efficiency for this domain. For larger domains, both the LHS and RHS perform reasonably well. On the largest domain, the SSOR simulated application achieved about 53% of the peak performance using eight processors of the Cray Y-MP.

[1]The inner loops access elements within a hyperplane and the outer loops iterate across the hyperplanes. See Sections 6.1.3 and 6.2.2 for a discussion about the parallelism and efficiency of the

Domain size	p	Memory (MBytes)	Time/step (seconds)	speedup	Efficiency (%)	Performance (Mflop/s)
$32 \times 32 \times 32$	1	10	0.254	-	-	191
	2	10	0.147	1.73	86.4	329
	4	10	0.087	2.92	73.0	557
	8	10	0.060	4.20	52.5	800
$64 \times 64 \times 64$	1	60	2.115	-	-	202
	2	60	1.124	1.88	94.1	381
	4	60	0.602	3.51	87.8	711
	8	60	0.345	6.13	76.7	1241
$102 \times 102 \times 102$	1	214	8.694	-	-	206
	2	214	4.551	1.91	95.6	394
	4	214	2.356	3.69	92.3	760
	8	214	1.292	6.73	84.1	1386
$128 \times 128 \times 128$	1	437	17.514	-	-	204
	2	438	9.074	1.93	96.5	395
	4	438	4.717	3.71	92.8	759
	8	438	2.530	6.92	86.5	1416

Table 1: Performance on the Cray Y-MP

Domain size	p	RHS		Lower		Upper		Time/step
		Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)
$32 \times 32 \times 32$	1	0.070	27.6	0.091	35.8	0.090	35.4	0.254
	4	0.020	22.6	0.034	38.4	0.033	38.1	0.087
	8	0.010	17.0	0.025	41.3	0.025	40.9	0.060
$64 \times 64 \times 64$	1	0.521	24.6	0.785	37.1	0.780	36.9	2.115
	4	0.140	23.3	0.228	37.9	0.227	37.7	0.602
	8	0.073	21.1	0.135	39.1	0.133	38.6	0.345
$102 \times 102 \times 102$	1	2.240	25.8	3.190	36.7	3.160	36.3	8.694
	4	0.581	24.7	0.877	37.2	0.871	37.0	2.356
	8	0.312	24.1	0.484	37.5	0.481	37.2	1.292
$128 \times 128 \times 128$	1	4.280	24.4	6.530	37.3	6.510	37.2	17.514
	4	1.129	23.9	1.771	37.6	1.766	37.4	4.717
	8	0.584	23.1	0.959	37.9	0.960	37.9	2.530

Table 2: Profile on the Cray Y-MP

Domain size	p	RHS (%)	LHS (%)	Time/step (%)
$32 \times 32 \times 32$	4	88.5	67.7	73.0
	8	85.0	45.5	52.5
$64 \times 64 \times 64$	4	93.0	86.0	87.8
	8	89.2	73.0	76.7
$102 \times 102 \times 102$	4	96.4	90.8	92.3
	8	89.7	82.3	84.1
$128 \times 128 \times 128$	4	94.8	92.2	92.8
	8	91.6	84.9	86.5

Table 3: Parallel efficiency of major parts of the code on the Cray Y-MP

6 DISTRIBUTED MEMORY IMPLEMENTATIONS

In this section, implementations of the SSOR simulated application on two distributed memory multiprocessors are presented. The machines used are a Thinking Machine CM-2 and an Intel iPSC/860. Both machines are located in the NAS Systems Division at NASA Ames. First, an approach to implementing the hyperplane algorithm, Algorithm 3.4, on distributed memory machines is presented and discussed.

6.1 HYPERPLANE ALGORITHM FOR DISTRIBUTED MEMORY MULTIPROCESSORS

Of the algorithms presented in Section 3.1, the hyperplane algorithm, Algorithm 3.4, exhibits the largest amount of concurrency. So, this section discusses one way to implement the hyperplane algorithm on distributed memory multiprocessors. For purposes of discussion, it is assumed that (a) an infinite number of processing elements (PEs) are available, (b) each PE has a microprocessor with cache, (c) each PE has a large amount of local memory, (d) PEs are connected in either a grid or a hypercube topology, and (e) communication with an adjacent processor is less costly than communication with a processor further away.

For distributed memory multiprocessors, an important question is how many processors an algorithm can use effectively. Given a cubic domain ($N \times N \times N$) and ignoring boundary conditions, the right hand side calculations (RHS) of (15) are completely parallel and can use $O(N^3)$ processors. The left hand side calculations (LHS) of (15) and (16) are limited to parallelism within a hyperplane, and so use at most $O(N^2)$ processors. Therefore, at most $O(N^2)$ processors are assumed.

Some desirable features of a parallel implementation of the hyperplane algorithm are the following:

- $O(N^2)$ processors are used effectively.
- Elements of a hyperplane are accessed efficiently.

hyperplane algorithm.

- Memory requirements of the parallel version are not significantly more than those of the serial version.

To use $O(N^2)$ processors effectively, a one- or two-dimensional partitioning of the domain is necessary. A three dimensional partitioning of the domain onto $O(N^2)$ processors where each partition is assigned to a separate processor will never use all of the processors since the parallelism occurs within hyperplanes. For one dimensional partitioning of the domain, the wavefront algorithm, Algorithm 3.3, can be used and so is not considered further.

Examining two dimensional partitionings of the domain, there are three ways to choose the two axes to partition. In a cubic domain, all three choices are equivalent. For noncubic domains, the two shortest axes are partitioned. When any hyperplane is projected onto the surface created by an axis pair, the ratio of the projected hyperplane to surface area is greatest for the two shortest axes. This implies a greater processor utilization if the two shortest axes are partitioned. Besides having the greatest processor utilization, parallelism within a hyperplane can never exceed the product of the two smallest dimensions.

Efficient access of elements in a hyperplane is difficult for microprocessors with a cache; there is a lack of spatial locality. If hyperplane elements are accessed along diagonals, elements are a large constant stride apart which causes cache misses. In addition, between the end of one diagonal and the beginning of the next is a large variable stride.

One possibility to increase data locality is to map hyperplanes into individual rectangular planes and then partition each rectangular plane across the processors. This eliminates the cache misses caused by the large stride. However, this approach "wastes" memory. For a cubic domain, about three times more memory is required since there are $(3N-2)$ hyperplanes and each is mapped to an $(N \times N)$ plane. For large problems, this is untenable.

If the hyperplanes could be packed into rectangular planes that occupy the original volume, then spatial locality would be enhanced and there would be no memory overhead. Such a solution is presented next.

6.1.1 SKEW HYPERPLANE MAPPING

The skew hyperplane mapping described below enhances data locality allowing efficient access to hyperplane elements and requires no extra memory. All hyperplanes are mapped onto planes normal to one of the coordinate axes. No extra memory is required since the mapped planes occupy the same volume as the original domain. In an actual parallel implementation, some memory may be required for communication buffers.

The definition of H_m given in the hyperplane algorithm does not include the boundary points. Here, H_m needs to be extended to include boundary points so the relationship between boundary points and interior points is maintained through the transformation. Therefore, the m-th hyperplane is defined by the set:

$$H_m = \{(i,j,k), |i+j+k=m\}, \quad m = 3,4,\cdots,(N_\xi + N_\eta + N_\zeta).$$

Figure 8: $H_m = 6$ hyperplane on $(5 \times 5 \times 5)$ domain.

Given that $N_\zeta \geq N_\xi$, and $N_\zeta \geq N_\eta$, there are at most $N_\xi N_\eta$ points in any hyperplane. This is seen by projecting the points of the hyperplanes onto the (ξ, η), (η, ζ) and (ξ, ζ) planes. If $N_\zeta \geq N_\xi + N_\eta - 1$, the projection onto the (ξ, η) plane fills the entire plane when $N_\xi + N_\eta + 1 \leq m \leq N_\zeta + 2$, whereas projections of a hyperplane never completely fill the (η, ζ) and (ξ, ζ) planes. Therefore, at most $P = N_\xi N_\eta$ processors should be used when the ξ, η dimensions are partitioned.

To pack hyperplanes into two dimensional planes that occupy the same volume as the original domain, a one-to-one mapping is required. This is accomplished by the following mapping:

$$\begin{aligned} i' &= i, \\ j' &= j, \text{ and} \\ k' &= ((i + j + k - 3) \; mod \; N_\zeta) + 1. \end{aligned}$$

Notice that when projecting a hyperplane onto the (ξ, η) plane only the ζ coordinate changes.

To see that the mapping is one-to-one, assume that two distinct points, (i_1, j_1, k_1) and (i_2, j_2, k_2), map to (i', j', k'). Since $i_1 = i_2 = i'$ and $j_1 = j_2 = j'$ then

$$k' = ((i' + j' + k_1 - 3) \; mod \; N_\zeta) + 1 = ((i' + j' + k_2 - 3) \; mod \; N_\zeta) + 1.$$

Without loss of generality, assume $k_1 < k_2$, then $k_2 = k_1 + cN_\zeta$ for some integer constant $c > 0$. However, this is a contradiction since $1 \leq k_1, k_2 \leq N_\zeta$. Therefore, the mapping is one-to-one.

In addition, at most three hyperplanes map to any given (ξ, η) plane. Since

$$\begin{aligned} 3 \leq m &\leq N_\xi + N_\eta + N_\zeta \\ &\leq 3N_\zeta \end{aligned}$$

and $k' = ((m-3) \; mod \; N_\zeta) + 1$, at most three hyperplanes can be mapped to any given k'.

6.1.2 INVERSE MAP FUNCTION

An inverse map function is necessary to determine how close a point is to the boundary where special processing is required. The ease of the inverse mapping depends upon whether the hyperplane number m is known or not. If it is known then

$$i = i', \; j = j', \text{ and } k = m - (i + j).$$

If the hyperplane number is not known then the inverse map is not as simple because of the *mod* function. The inverse mapping is given by:

$$i = i', \; j = j', \text{ and } k = \begin{cases} k' - ((i+j-2) \; mod \; N_\zeta) & \text{if } k' > (i+j-2) \; mod \; N_\zeta \\ N_\zeta + k' - ((i+j-2) \; mod \; N_\zeta) & \text{if } k' \leq (i+j-2) \; mod \; N_\zeta. \end{cases}$$

6.1.3 DEGREE OF PARALLELISM

Parallelism within a hyperplane is equal to the number of points in the hyperplane. Below, a general expression for the amount of parallelism in the m-th hyperplane is presented.

In addition to the assumptions presented in Section 6.1.1, it is assumed that $N_\zeta \geq N_\eta \geq N_\xi$. Then the expression for parallelism in the m-th hyperplane is given by the following formula:

$$P(H_m) = \begin{cases} \frac{1}{2}(m-2)(m-1) & \text{if } 3 \leq m \leq N_\xi + 2 \\ N_\xi N_\eta + \text{ appropriate term(s) from Table 4} & \text{if } N_\xi + 3 \leq m \leq N_\eta + N_\zeta + 1 \\ \frac{1}{2}(N_\xi + N_\eta + N_\zeta - m + 1)(N_\xi + N_\eta + N_\zeta - m + 2) & \text{if } N_\eta + N_\zeta + 2 \leq m \leq N_\xi + N_\eta + N_\zeta, \end{cases}$$

where $P(H_m)$ is the amount of parallelism within the m-th hyperplane. For hyperplanes in the range $N_\xi + 3 \leq m \leq N_\eta + N_\zeta + 1$, the expression for parallelism is formed by taking $N_\xi N_\eta$ and adding all appropriate terms from Table 4. Depending upon the actual dimensions of the domain, some of the terms may not apply.

As an example of using Table 4, let $N_\xi = 4$, $N_\eta = 6$ and $N_\zeta = 7$. Then for hyperplane $m = 10$,

$$\begin{aligned} P(H_{10}) &= (4)(6) - \tfrac{1}{2}(4+6-10+1)(4+6-10+2) \\ &\qquad\quad - \tfrac{1}{2}(10-7-2)(10-7-1) \\ &= 22 \end{aligned}$$

using lines two and three of Table 4.

Term	Condition
$-\frac{1}{2}(N_\xi + 2N_\eta - 2m + 3)N_\xi$	$N_\xi + 3 \quad \leq m \leq \quad N_\eta + 2$
$-\frac{1}{2}(N_\xi + N_\eta - m + 1)(N_\xi + N_\eta - m + 2)$	$N_\eta + 3 \quad \leq m \leq \quad N_\xi + N_\eta + 1$
$-\frac{1}{2}(m - N_\zeta - 2)(m - N_\zeta - 1)$	$N_\zeta + 3 \quad \leq m \leq \quad N_\xi + N_\zeta + 1$
$-\frac{1}{2}(2m - N_\xi - 2N_\zeta - 3)N_\xi$	$N_\xi + N_\zeta + 2 \quad \leq m \leq \quad N_\eta + N_\zeta + 1$

Table 4: Terms used to calculate parallelism for the m-th hyperplane.

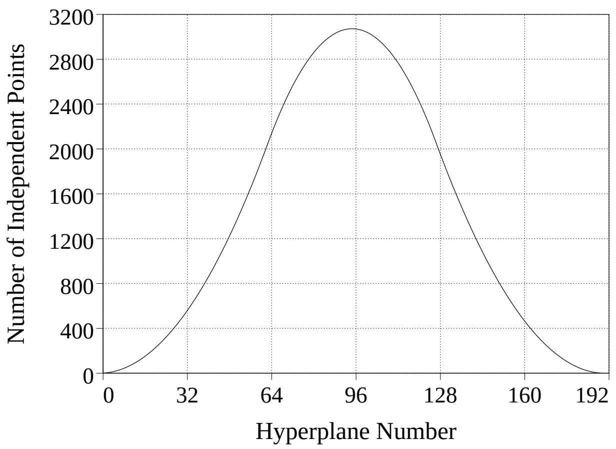

Figure 9: Parallelism within hyperplanes for a $(64 \times 64 \times 64)$ domain.

For a cubic domain $(N \times N \times N)$ the equations for parallelism reduce to the following:

$$P(H_m) = \begin{cases} \frac{1}{2}(m-2)(m-1) & \text{if } 3 \leq m \leq N+2 \\ N^2 \quad -\frac{1}{2}(2N-m+1)(2N-m+2) & \\ \qquad -\frac{1}{2}(m-N-2)(m-N-1) & \text{if } N+3 \leq m \leq 2N+1 \\ \frac{1}{2}(3N-m+1)(3N-m+2) & \text{if } 2N+2 \leq m \leq 3N. \end{cases}$$

These equations are equivalent to the ones reported in [3] for cubic domains. The maximum parallelism within any single hyperplane is found to be $(\frac{3}{4}N^2 + \frac{1}{4})$ which matches the value reported in [3].

It should be noted that for a cubic domain, no hyperplane can fully utilize $O(N^2)$ processors. Figure 9 shows the amount of parallelism within each hyperplane for a $(64 \times 64 \times 64)$ domain. The maximum parallelism is 3072 which is less than 4096 (64^2).

6.1.4 USING THE SKEW HYPERPLANE MAPPING

Two ways to use the skew hyperplane mapping are to either (i) apply the skew hyperplane mapping every time step before computing the LHS and then use the inverse mapping before computing the RHS or (ii) apply the skew hyperplane mapping once and operate completely within the transformed domain when computing the LHS and RHS. Mapping and unmapping the computational domain when the LHS is computed, will leave the RHS calculation alone to operate on the unmapped domain. Mapping the domain once at the outset avoids the cost of mapping and unmapping the domain every time step. Unfortunately, mapping the domain once at the outset makes the RHS computation more difficult because the original ζ boundaries have become hyperplanes in the mapped domain. However, the penalty for making the RHS more difficult may be offset by avoiding the cost of mapping and unmapping every time step.

For the RHS, 2nd and 4th order differences are computed which require information from nearest neighbor and next to nearest neighbor grid points. With a two dimensional partitioning and the skew hyperplane mapping, all interprocessor communication is nearest neighbor or next to nearest neighbor. All communication will be nearest neighbor if at least two points in the ξ and η dimensions are assigned to each processor.

6.2 EFFICIENCY CONSIDERATIONS

In this section, efficiency limits of the wavefront algorithm, Algorithm 3.3, using a one dimensional partitioning and the hyperplane algorithm, Algorithm 3.4, using the skew hyperplane mapping are examined for $(N \times N \times N)$ cubic domains. It is assumed that communication costs are zero and that boundary elements are included in the computation and treated as interior elements.

6.2.1 WAVEFRONT ALGORITHM EFFICIENCY

For the wavefront algorithm, $O(N)$ processors are assumed because mesh diagonals have $O(N)$ parallelism. Given that the first dimension of the $(N \times N \times N)$ cubic domain is partitioned using $P \leq N$ processors, each subdomain local to a processor is $(N_l \times N \times N)$, where $N_l = N/P$.

Ignoring special processing at the boundaries, execution time on one processor is assumed to be directly proportional to the number of grid points, i.e., $T_1 \propto N^3$. Execution time on P processors is proportional to the amount of work done along the critical path. The critical path is the longest path of execution and may cross processor boundaries. For a fixed value of k, work within the plane is partitioned into a completely concurrent part where updates are made from the previous k plane and a partially concurrent part where updates are done in parallel within a mesh diagonal but the mesh diagonals are handled sequentially, see Algorithm 3.3. Approximately one fourth of the work within a k plane occurs in the update from the previous plane and about three fourths when operating on mesh diagonals. So, the execution time on P processors is approximated by:

$$T_P \propto \frac{1}{4}N^2 N_l + \frac{3}{4}N((2N-1)N_l - 2(N_l-1)N_l/2).$$

where $\frac{1}{4}N^2 N_l$ is proportional to the work done to update all points from the previous k plane and $\frac{3}{4}N((2N-1)N_l - 2(N_l-1)N_l/2$ is proportional to the work done on mesh diagonals within the k plane. The first term in the mesh diagonal work implies there is always one processor that has a full amount of work, $(N_l N)$ grid points, for each of the $(2N-1)$ mesh diagonals. The last term is a correction that accounts for the time required for the first processor to fill and for the last processor to empty. There are $(N_l - 1)$ mesh diagonals processed before the first processor becomes full and each mesh diagonal has an average of $\frac{N_l N}{2}$ grid points. Therefore, speedup is equal to the following expression:

$$\begin{aligned} S_W &= \frac{N^3}{\frac{1}{4}N^2 N_l + \frac{3}{4}N((2N-1)N_l - 2(N_l-1)N_l/2)} \\ &= \frac{4P}{7-3/P}. \end{aligned} \tag{18}$$

Substituting for the speedup in the formula for efficiency yields:

$$\begin{aligned} e_W &= \frac{\frac{4P}{7-3/P}}{P} \\ &= \frac{4}{7-3/P}. \end{aligned}$$

This implies, efficiency of the wavefront algorithm when using $O(N)$ processors approaches $\frac{4}{7}$ for cubic domains. This compares well with results reported in by Greenbaum in [7] where for two dimensional triangular solves, an efficiency of $\frac{1}{2}$ is reported. Here efficiency is greater than $\frac{1}{2}$ because the update using the previous k plane is done completely in parallel and so raises the efficiency.

6.2.2 HYPERPLANE ALGORITHM EFFICIENCY

For the hyperplane algorithm, $O(N^2)$ processors are assumed since hyperplanes have at most $O(N^2)$ parallelism. Given that the first two dimensions of the $(N \times N \times N)$ cubic domain are partitioned using $P \leq N^2$ processors in a $(\sqrt{P} \times \sqrt{P})$ processor grid, each subdomain local to a processor is $(N_l \times N_l \times N)$, where $N_l = N/\sqrt{P}$.

Ignoring special processing at the boundaries, the execution time on one processor is assumed to be directly proportional to the number of grid points, i.e., $T_1 \propto N^3$. Execution time on P processors is proportional to the amount of work done along the critical path through the hyperplanes which is proportional to the number of grid points in the processors along the critical path. So, the execution time on P processors is approximated by:

$$T_P \propto (3N-2)N_l^2 - 2(2N_l - 2)N_l^2/2,$$

where $(3N-2)$ are the number of hyperplanes, N_l^2 is the local work and the last term accounts for the fact that not all hyperplanes have N_l^2 grid points, $(2N_l - 2)$ hyperplanes, each with and average of $N_l^2/2$ points during startup and draining of the pipeline. Therefore, speedup is equal to the following expression:

$$\begin{aligned} S_H &= \frac{N^3}{(3N-2)N_l^2 - 2(2N_l-2)N_l^2/2} \\ &= \frac{P}{3-2/\sqrt{P}}. \end{aligned} \tag{19}$$

Substituting for the speedup in the formula for efficiency yields:

$$\begin{aligned} e_H &= \frac{\frac{P}{3-2/\sqrt{P}}}{P} \\ &= \frac{1}{3-2/\sqrt{P}}, \end{aligned}$$

or approximately $\frac{1}{3}$. This implies, efficiency of the hyperplane algorithm when using $O(N^2)$ processors approaches $\frac{1}{3}$ for cubic domains.

6.2.3 EFFICIENCY LIMIT COMPARISON

Efficiency limits presented in the previous sections, Sections 6.2.1 and 6.2.2, demonstrate the trade-off between the number of processors and algorithm. The wavefront algorithm, Algorithm 3.3, has better efficiency than the hyperplane algorithm, Algorithm 3.4, when the number of processors is less than N. However, when the number of processors is larger than N, efficiency of the wavefront algorithm decreases rapidly until the hyperplane algorithm becomes more efficient.

For $N \leq P \leq N^2$, the efficiencies are

$$e_W = \frac{4}{7-3/N}\frac{N}{P}$$

and

$$e_H = \frac{1}{3-2/\sqrt{P}},$$

where P is the number of processors, N is one grid dimension, and e_W and e_H are efficiencies for the wavefront and diagonal hyperplane algorithms respectively. The cross over point approaches $P = \frac{12}{7}N$ as N and P approach infinity. Figure 10 demonstrates the cross over of efficiencies on a $(64 \times 64 \times 64)$ domain. Note that the cross over point in Figure 10 occurs near $1.61N$ rather than $1.71N$.

All of the efficiency calculations are approximations and communication has not been taken into account. However, they do indicate that on an ideal parallel machine, the number of processors and the size of the computational domain must be considered when selecting an algorithm.

For an actual application, communication costs and other phases of the problem will change the efficiencies. The RHS calculations are fully parallel and should raise the overall efficiency. On the other hand, if interprocessor communication is considered, it is expected that both the wavefront and hyperplane efficiencies will be less than predicted. However, the wavefront efficiency should degrade more than the hyperplane efficiency because the surface to volume ratio is worse for one dimensional partitionings than two dimensional partitionings and communication is proportional to surface area.

Another phase that has not been considered is the formation of the LHS. For the hyperplane algorithm, the LHS can be formed in parallel but there might not be enough memory. The LHS can be formed for the current hyperplane as needed, implying there will be $3N-2$ stages where the LHS is being formed. For the wavefront

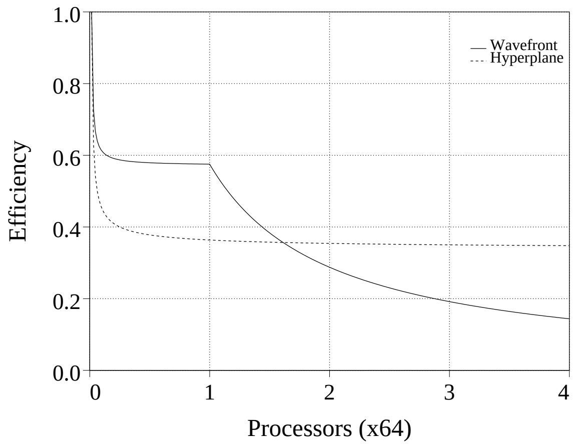

Figure 10: Efficiency curves for the wavefront and hyperplane algorithms on a $(64 \times 64 \times 64)$ domain.

algorithm, the LHS can be formed a k plane at a time, implying N stages where the LHS is being formed. This will make the wavefront implementation more efficient relative to the hyperplane implementation.

Finally, it should be noted that cubic domains have worse efficiency limits than noncubic domains.

6.3 CM2 IMPLEMENTATION

The model problem, Symmetric Successive Over-Relaxation simulated application, has been implemented on the Connection Machine (CM2) at NASA Ames using the skew hyperplane mapping approach.

6.3.1 CM2 SYSTEM DESCRIPTION

The CM2 has 32k 1-bit serial processors, 1024 64-bit Weitek floating point units (FPU), 4 GBytes of memory, a clock rate of 7 MHz, and a Sun 4/490 as a front end machine. The peak performance of the machine is 14 Gflop/s.

The Connection Machine Fortran (CMF) compiler has two execution models, fieldwise and slicewise. The fieldwise model uses both the single bit processors and the FPUs, the single bit processor for integer and logical operations and the FPUs for floating point operations. The slicewise model uses only the FPUs for all arithmetic and logical operations. The implementation described in this section only uses the slicewise model. For information on optimizing within the slicewise model see [11].

In the slicewise model, the CM2 at NASA Ames is considered a SIMD machine with 1024 processing elements (PEs). Each PE is a vertex of a ten dimensional hypercube with a FPU and 4 MBytes of memory. Compiler directives allow users to specify whether an individual axis of an array as either parallel, spread across processors, or serial, contained within a processor. The CMF compiler partitions an array with parallel dimensions into subgrids based on the array size and the machine size. All subgrids are the same size: the compiler pads an array as required to form equal sized subgrids. Each subgrid is assigned to a PE. Besides requiring all subgrids to be the same size, the size is required to be a multiple of four since the FPU has a four stage pipeline. If the number of grid points in the parallel dimensions is not a multiple of four times the number of FPUs, the compiler pads the array.

6.3.2 CM2 IMPLEMENTATION DETAILS

The hyperplane algorithm, Algorithm 3.4, is implemented on the CM2 using the skew hyperplane mapping approach. The domain is partitioned by distributing the N_ξ and N_η dimensions among processors and storing the N_ζ dimension in-processor. The CMF compiler maps the parallel dimensions onto the CM array first, and then the serial dimension is added on top of this mapping. This means that the domain is partitioned into pencils where the serial dimension of the subgrids is fixed, independent of the mapping.

The skew hyperplane mapping approach requires the computational domain be transformed. In this implementation, the computational domain is transformed once at the beginning of the computation and both the RHS and LHS operate in the transformed domain. The RHS computation operates on all grid points of the transformed domain simultaneously since there are no data dependencies between hyperplanes. The LHS computation is done sequentially across hyperplanes since there are dependencies between hyperplanes.

The computation of the fourth-difference dissipative terms near the computational boundaries requires special attention. For the two planes next to the boundaries, equations 8 through 11 are used to compute the fourth-difference dissipative terms. All other interior planes use equation 7. A straightforward CM Fortran implementation requires a boolean mask for every different plane type. This means that for the $\xi-$ direction four 2-D masks are needed for $i = 2, 3, N_\xi - 2$, and $N_\xi - 1$ and a 3-D mask for $4 \leq i \leq N_\xi - 3$. A total of 15 masks are needed for the three directions. These masks are precalculated and used every time step. However, this implementation has a deficiency. Since the CM2 is a SIMD machine, all processors execute one instruction every cycle, so, operating on one plane of a 3-D array means other planes will be unaffected, i.e., processors associated to these planes will be idle.

An alternative approach is based on the observation that the coefficients of U^n terms in equations 8 through 11 and equation 7 are similar. For example, the $U^n_{i+1,j,k}$ term has the same coefficient (-4) in the four equations that it appears. This is also true for the coefficients of $U^n_{i+2,j,k}$, $U^n_{i-1,j,k}$, and $U^n_{i-2,j,k}$ terms. Only $U^n_{i,j,k}$ has a coefficient of either 5 or 6. Based on this observation, each U^n term can be computed separately from the others; i.e., $U^n_{i+2,j,k}$ is computed first for $2 \leq i \leq N_\xi - 3$, then $U^n_{i+1,j,k}$ is computed for $2 \leq i \leq N_\xi - 2$, and so on. This implementation would require a 3-D boolean mask for every term in every direction. A total of 15 masks are required for the three directions. ($U^n_{i,j,k}$ can be computed for all planes with a coefficient of 6 and recomputed for $i = 2$ and $\mathrm{N}_\xi - 1$ with a coefficient of -1.) This implementation is more efficient than the previous one since most processors will be active most of the time.

A third approach for computing the dissipative terms is based on the observation that all terms of equations 8 through 11 and equation 7 have constant coefficients.

These coefficients can be precalculated and stored in 3-D arrays. Array elements corresponding to the missing terms are set to zero. One 3-D array is needed for every U^n term in every direction. The coefficient arrays can be either real or integer (since the coefficients are integers and reals are stored as 64-bit numbers, 32-bit integers can be used to save memory). Again, a total of 15 arrays are needed for the three directions. In this approach, as in the second one, most processors will be active most of time.

The choice between the three approaches depends on many factors including performance and storage requirements. The first approach requires the least amount of memory, especially if boolean variables are represented by single bits. However, this approach is the least efficient one in terms of processor utilization. The three approaches have been implemented on the CM2, and the third approach found to be significantly faster than the first and slightly better than the second. With respect to memory requirements, both integer and boolean variables, in the slicewise model, are represented by 32 bits, so, the second and third approaches require the same amount of memory. Since memory requirements are the same, the third method is used.

The LHS computation also requires a 3-D boolean mask since multiple hyperplanes are mapped to a single rectangular plane. This mask has $N_\xi + N_\eta + N_\zeta - 2$ planes, one for every hyperplane. However, conditional store operations (mask operations) on the CM2 are not as efficient as the unconditional store operations. In fact, conditional stores are about three times slower than unconditional stores. Therefore, it is more efficient to perform the LHS computation on all grid points in every plane and only mask out the unneeded points when computing the residuals at the end of each triangular solve.

6.3.3 CM2 RESULTS

The results of implementing the SSOR simulated application for several domain sizes on the CM2 are listed in Table 5. The CMF compiler release 1.1 operating in the slicewise mode was used to generate these results. All arithmetic is done in double precision (64 bit). Notice that the number of PEs is based on the 64-bit floating point processors rather than the 1-bit serial processors. Timing, in seconds, are for a single time step averaged over ten steps. Performance rates are based on these timings and the number of flops computed on the Cray Y-MP, see Section 4.

Table 6 shows the subgrid size and memory requirement that the CMF compiler uses to map the various problem sizes onto the CM2. The compiler partitions the parallel dimensions at run time to generate the optimal subgrid size for the number of processors used. Memory requirements given in the table are based on the programmer's declared variables, both local and global. Temporary variables created by the compiler are not included. Notice that for the non-power-of-two domains the required memory depends on the number of processors since the compiler uses different padding schemes to partition the domain into equal sized subgrids.

Results for the smallest grid, $(32 \times 32 \times 32)$, show no performance improvement when more than 256 PEs are used. This means that only a portion of the machine, 256 PEs, is utilized even when more processors are available.

The non-power-of-two grid, $(102 \times 102 \times 102)$, has two implementations depending on the shift operation used. CM Fortran has two shift functions: *cshift* (circular shift) and *eoshift* (end-off shift). The cshift function shifts data in one direction by a user specifiable distance. Data shifted off one end of the array is inserted at the opposite edge. The eoshift function is similar to cshift except that values inserted are user specifiable constants (default is zero) instead of the values shifted out. The current implementation of shift operations favors cshift for power-of-two arrays and eoshift for non-power-of-two arrays. Results (A) for the $(102 \times 102 \times 102)$ domain, see Tables 5 and 6, are obtained by declaring all arrays as $(102 \times 102 \times 102)$ and using eoshift. Results (B) for the $(102 \times 102 \times 102)$ domain are obtained by declaring arrays to be $(128 \times 128 \times 102)$ and using cshift in most places and eoshift with array sections in other places. The results show that implementation (B) is more efficient than implementation (A) even though there are more operations.

Table 5 shows performance improvement for the power-of-two grid sizes as the number of processors is increased. Results for the non-power-of-two domains are less impressive because of the padding done by the compiler and the implementation of shift operations.

Costs of the main computational parts of the SSOR simulated application are given in Table 7. The main computational parts are the RHS, lower triangular solve, and upper triangular solve. Together, they represent about 99% of the execution time within a time step. For the power-of-two domains, the RHS computation contributes only 20% to 23% to the total time while the LHS computation contributes 76% to 79% to the total time. For the $(102 \times 102 \times 102)$ domain, the RHS computation is about 26% of the total time when using eoshifts, implementation (A), and about 35% of the total time when using cshifts, implementation (B). For implementation (B), eoshift with array sections are used in the RHS computation.

6.3.4 ANALYSIS OF CM2 RESULTS

The skew hyperplane mapping is very efficient on the CM2. Since at least two grid points in every parallel dimension are assigned to every PE, see Table 6, communication is strictly nearest neighbor. The only deficiency of the skew hyperplane mapping is processor idling during the LHS computation. This is inherent to this scheme since there is a data dependency between hyperplanes. On average, only one third of the processors are performing useful work for the LHS computation.

In order to estimate the communication overhead in the SSOR simulated application, the program is run with all shift functions removed (substituted by array elements that are not shifted). Results are considered to be for computation only. The difference between the computation only time and the actual time is the communication overhead. This procedure approximates the communication overhead since the compiler may do some optimization that is prevented in the presence of shift operations. Table 8 lists the estimated computation and communication times for several domains on the CM2. For power-of-two domains, the communication time depends primarily on the subgrid size. The times are relatively low for large subgrids, 27% of the execution time for the $(8 \times 8 \times 128)$ subgrid, and relatively high for small subgrids, 58% for the $(2 \times 2 \times 64)$ subgrid. For the non-power-of-two domains, the communication time also depends on the type of shift operation. Results

Domain size	PEs	Time/step (sec)	Rate (Mflop/s)
$32 \times 32 \times 32$	256	1.140	42
	512	1.121	43
	1024	1.124	43
$64 \times 64 \times 64$	256	5.226	82
	512	3.400	125
	1024	2.288	186
$102 \times 102 \times 102$ (A)	256	37.786	47
	512	22.081	81
	1024	13.353	134
$102 \times 102 \times 102$ (B)	256	29.867	60
	512	17.009	105
	1024	10.012	179
$128 \times 128 \times 128$	256	31.429	114
	512	18.242	196
	1024	10.724	333

Table 5: Performance on the CM2

Domain size	PEs	Subgrid size	Memory (MBytes)
$32 \times 32 \times 32$	256	$2 \times 2 \times 32$	14
	512	$2 \times 2 \times 32$	14
	1024	$2 \times 2 \times 32$	14
$64 \times 64 \times 64$	256	$4 \times 4 \times 64$	113
	512	$4 \times 2 \times 64$	113
	1024	$2 \times 2 \times 64$	113
$102 \times 102 \times 102$ (A)	256	$26 \times 2 \times 102$	576
	512	$14 \times 2 \times 102$	620
	1024	$4 \times 4 \times 102$	709
$102 \times 102 \times 102$ (B)	256	$8 \times 8 \times 102$	709
	512	$8 \times 4 \times 102$	709
	1024	$4 \times 4 \times 102$	709
$128 \times 128 \times 128$	256	$8 \times 8 \times 128$	886
	512	$8 \times 4 \times 128$	886
	1024	$4 \times 4 \times 128$	886

Table 6: Partitioning on the CM2

Domain size	PEs	RHS		Lower		Upper		Time/step
		Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)
$32 \times 32 \times 32$	256	0.245	21.5	0.442	38.8	0.437	38.3	1.140
	512	0.244	21.8	0.433	38.6	0.433	38.6	1.121
	1024	0.243	21.6	0.436	38.8	0.435	38.7	1.124
$64 \times 64 \times 64$	256	1.192	22.8	2.000	38.3	1.988	38.0	5.226
	512	0.765	22.5	1.312	38.6	1.299	38.2	3.400
	1024	0.465	20.3	0.910	39.8	0.893	39.0	2.288
$102 \times 102 \times 102$ (A)	256	10.124	26.8	13.981	37.0	13.607	36.0	37.786
	512	5.808	26.3	8.248	37.4	7.960	36.0	22.081
	1024	3.383	25.3	5.081	35.1	4.863	36.4	13.353
$102 \times 102 \times 102$ (B)	256	10.305	34.5	9.774	32.7	9.694	32.5	29.867
	512	5.885	34.6	5.580	32.8	5.494	32.3	17.009
	1024	3.480	34.8	3.266	32.6	3.236	32.3	10.012
$128 \times 128 \times 128$	256	6.214	19.8	12.295	39.1	12.064	38.4	31.429
	512	3.888	21.3	6.997	38.4	6.920	37.9	18.242
	1024	2.379	22.2	4.077	38.0	4.050	37.8	10.724

Table 7: Profile on the CM2

Domain size	PEs	Computation		Communication		Time/step
		Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)
$32 \times 32 \times 32$	256	0.488	42.8	0.652	57.2	1.140
$64 \times 64 \times 64$	256	3.102	59.4	2.124	40.6	5.226
	512	1.685	49.6	1.715	50.4	3.400
	1024	0.957	41.8	1.331	58.2	2.288
$102 \times 102 \times 102$ (A)	256	14.827	39.2	22.959	60.8	37.786
	512	8.165	37.0	13.916	63.0	22.081
	1024	4.839	36.2	8.514	63.8	13.353
$102 \times 102 \times 102$ (B)	256	18.955	63.5	10.912	36.5	29.867
	512	9.991	58.7	7.018	41.3	17.009
	1024	4.847	48.4	5.165	51.6	10.012
$128 \times 128 \times 128$	256	22.962	73.1	8.467	26.9	31.429
	512	11.689	64.1	6.553	35.9	18.242
	1024	6.092	56.8	4.632	43.2	10.724

Table 8: Communication cost on the CM2

show that shift operations are more expensive for non-power-of-two arrays than for power-of-two arrays.

One conclusion drawn from these results is that performance variations for different domain/machine configurations are largely due to the communication cost. The computational time per grid point per processor (computed by dividing the computation cost by the subgrid size) is about the same for all cases, roughly 3 milliseconds. This means that the falloff from ideal speedup is primarily due to communication. Also, the communication time is significant for all domains even though the communication is only nearest neighbor.

6.4 iPSC/860 IMPLEMENTATION

In this section, four different implementations of the SSOR simulated application on the 128 node Intel iPSC/860 located at the NAS systems division at NASA Ames are presented. Two of the implementations are based on the wavefront algorithm, Algorithm 3.3. The other two implementations are based on the hyperplane algorithm, Algorithm 3.4.

6.4.1 iPSC/860 SYSTEM DESCRIPTION

The Intel iPSC/860 is a Multiple Instruction Multiple Data stream (MIMD) parallel computer [9]. The machine used has 128 processor nodes. Each node is comprised of a 40 MHz Intel i860 micro-processor, 8 MBytes of memory and a link to the hypercube communication network. Each node has a peak performance of 60 Mflop/s in 64-bit arithmetic.

6.4.2 iPSC/860 IMPLEMENTATION DETAILS

One of the main issues to be addressed in the implementation on a MIMD computer is the partitioning of the domain. We choose to keep the ζ dimension in-processor resulting in 2-D parallelism. Therefore, the sub-domains consist of pencils with the ζ direction being the in-processor direction. This limits the number of processors that can be used to solve a problem of a given size since only the points in the ξ and η directions can be distributed. The first partitioning strategy divides the $\xi\eta$ plane into blocks and is referred to as 2-D partitioning. Figure 11 illustrates this for 16 processors. Note that 1-D partitioning, where both η and ζ indices are held in-processor, is a subset of the 2-D partitioning. In this case, the sub-domains consist of slabs. In Section 6.4.5, other partitioning strategies are introduced.

Since the RHS calculation uses a 13 point stencil, an overlap region two cells wide between the partitions is used to facilitate the interprocessor communication. This increases the memory requirement. Other sources of extra memory are pointers for accessing hyperplanes, communication buffers, etc.

Two algorithms, namely the wavefront algorithm, Algorithm 3.3, and the hyperplane algorithm, Algorithm 3.4, have been implemented. We first discuss the implementation of the wavefront algorithm and present results for various problem sizes.

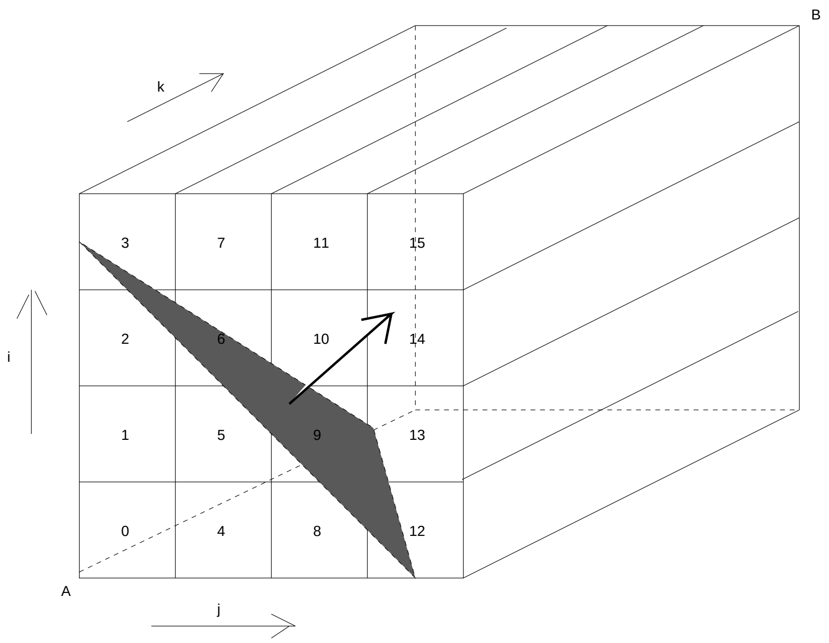

Figure 11: Two-dimensional partitioning with 16 processors; also shown are a hyperplane and the direction of access of hyperlanes for lower triangular solve.

6.4.3 iPSC/860 WAVEFRONT IMPLEMENTATION

The wavefront algorithm is straight-forward to implement. Referring to Figure 5 and Algorithm 3.3 and assuming a 1-D partitioning, the main steps of the algorithm are:

1. Compute the RHS. This step requires that each processor exchange two planes of data to its neighboring processors in the ξ and η directions.

2. Carry out the lower triangular solve by stepping through N_ζ. For each k, form the Jacobian matrices in $A_{i,j,k}$, $B_{i,j,k}$, $C_{i,j,k}$ and $D_{i,j,k}$, compute the contribution from $C_{i,j,k}$. Then step through the mesh diagonals to compute the rest of the contributions. Computation on a mesh diagonal requires communication whenever the diagonal crosses a processor boundary.

3. Carry out the upper triangular solve. This step is identical to step 2 except that the directions of access of N_ζ and of the mesh diagonals are reversed.

Since the last phase of the triangular solves using the wavefront algorithm only possesses 1-D parallelism, a 1-D partitioning seems natural. However, both 1-D and 2-D partitionings are implemented. It would appear that the 1-D partitioning should yield better results than the 2-D partitioning since it has better processor utilization during the triangular solves. However, this is not so as evidenced by the results presented in the following section.

A lower bound on the number of processors for a given domain is determined by the memory requirements for the implementation. Approximately 35,000 grid points fit in single processor. The grid points can be distributed in the ξ and η directions in any manner as long as the entire ζ dimension is held in-processor. There is also an upper bound on the number of processors since we require a minimum of 4 grid points in each of the ξ and η directions to facilitate the implementation. Since the RHS stencil is 5 points wide in each direction, a minimum of 2 points should suffice to retain nearest neighbor communication. However, it is easier to implement the dissipative terms near the computational boundaries if 4 grid points are used.

6.4.4 iPSC/860 WAVEFRONT RESULTS

In this section, results are presented for the SSOR simulated application using the wavefront algorithm on various domains. In all instances, execution time is the average time taken for a single time step of the computation in seconds.

Results from the 1-D and 2-D partitioning strategies are shown in Tables 9 and 10, respectively. The 1-D partitioning imposes severe restrictions on the domains that can be handled. For the domains that can be accommodated, 1-D partitioning has worse performance than 2-D partitioning. This is surprising since the the triangular solves possess only 1-D parallelism which should make the 1-D partitioning more suitable.

The reason for the poor performance of the 1-D partitioning is the 2-D partitioning's ability to overlap the Jacobian computations from the next k plane with the current plane. When the triangular solves are timed independently of the Jacobian calculations, the 1-D partitioning is faster. This effect does not appear in [7] because it is a result of the three dimensionality of the SSOR simulated application.

Problem size	No. of processors							
	1	2	4	8	16	32	64	128
32^3	10.51	5.84	3.64	2.37	-	-	-	-
64^3	-	-	-	-	10.45	-	-	-
102^3	-	-	-	-	-	-	-	-
128^3	-	-	-	-	-	-	-	-

Table 9: Elapsed time/step in secs. using wavefront algorithm on the Intel iPSC/860 using 1-D partitioning.

In Table 11, memory requirements and profile traces for the wavefront algorithm with a 2-D partitioning are presented using 64 and 128 processors for various domains. Times are presented for the RHS computation and the lower and upper triangular solves which comprise more than 99% of the work involved at each time step. Over a range of domains, the triangular solves comprise nearly 80-82% of the time.

6.4.5 iPSC/860 HYPERPLANE IMPLEMENTATION

Figure 11, which also shows a 2-D partitioning, contains a a hyperplane and the direction in which hyperplanes are accessed for the lower triangular solve. At each time step, the work involved consists of a lower triangular solve, an upper triangular solve and a RHS evaluation. The two triangular solves proceed in a sequential manner,

Problem size	No. of processors							
	1	2	4	8	16	32	64	128
32^3	10.51	5.84	3.15	2.00	1.20	0.76	0.46	-
64^3	-	-	-	-	8.27	4.74	2.74	1.72
102^3	-	-	-	-	-	-	9.93	6.14
128^3	-	-	-	-	-	-	-	10.56

Table 10: Elapsed time/step in secs. using wavefront algorithm on the Intel iPSC/860 using 2-D partitioning.

Problem size	p	RHS		Lower		Upper		Time/step	Memory
		Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)	(MBytes)
32^3	64	0.087	18.0	0.198	41.0	0.198	41.0	0.483	28
64^3	64	0.521	18.9	1.121	40.6	1.121	40.6	2.763	94
	128	0.310	14.0	0.730	43.3	0.730	42.1	1.770	130
102^3	64	1.882	18.9	4.078	40.9	4.003	40.2	9.963	290
	128	1.131	17.1	2.796	42.4	2.670	40.5	6.597	376
128^3	128	1.968	17.8	4.560	41.3	4.510	40.9	14.750	599

Table 11: Profile with wavefront algorithm.

starting at one corner of the computational domain and traverse along hyperplanes to the diametrically opposite corner of the domain. The amount of parallelism (the size of the hyperplane) varies as derived in Section 6.1.3. The hyperplane is defined by using appropriate data structures. Figure 12 illustrates the effect of the mapping on a $(4 \times 4 \times 4)$ computational domain. It shows the hyperplane numbers as a function of the position (i, j, k') and these are obtained by projecting the hyperplanes onto the $\xi\eta$ plane as described in Section 6.1.1. It is seen that each hyperplane is delimited by two diagonal wavefronts, a 'leading edge' and a 'trailing edge'. It is further seen that for a fixed k', we have no more than three different hyperplanes.

The following discussion pertains to the lower triangular solve and carries over to upper triangular solve with obvious modifications. The grid points (i, j, k), $(i-1, j, k)$,$(i, j-1, k)$ and $(i, j, k-1)$ are mapped under the skew hyperplane mapping onto (i, j, k'), $(i-1, j, (k-1)')$, $(i, j-1, (k-1)')$ and $(i, j, (k-1)')$. No communication is required for the contribution from the grid point $(i, j, k-1)$ since the ζ index is held in-processor. Communication is needed for the contributions from $(i-1, j, k)$ and $(i, j-1, k)$ whenever a hyperplane crosses a processor boundary. The processor receives $(i-1, j, k)$ from its south neighbor and $(i, j-1, k)$ from its west neighbor. Thus, each processor in the hyperplane receives boundary data from its south and west neighbors; it performs the computation, and sends boundary data, to its north and east neighbors.

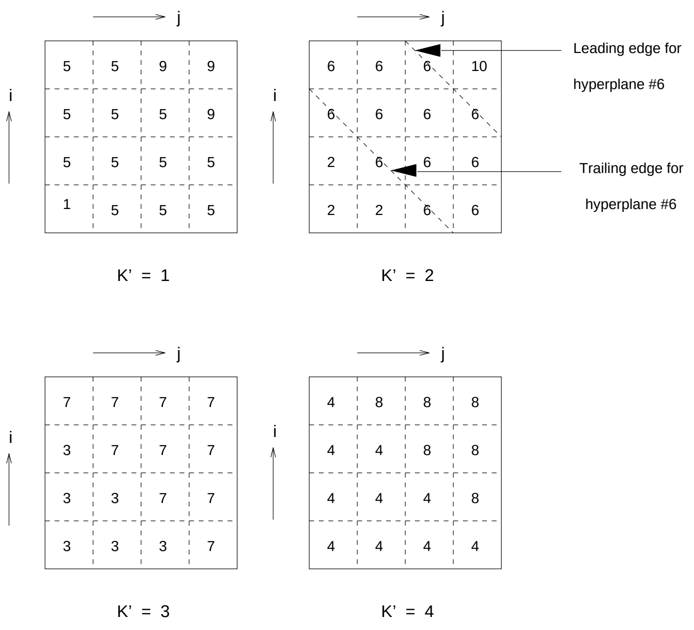

Figure 12: Skew hyperplane mapping of a $(4 \times 4 \times 4)$ computational domain problem with hyperplane numbers indicated.

For the RHS computation, grid point (i, j, k) requires information from 12 grid points that are distance 1 and distance 2 away along each dimension. No communication is required for the 4 grid points ζ direction since the ζ dimension is held in-processor. Two strategies can be employed for communication in the ξ direction and in the η direction.

In the first approach, the RHS computation follows the same path as a lower triangular solve; i.e. each processor receives data consisting of 8 boundary lines (constant ξ, ζ or η, ζ lines) from its four neighbors, computes and sends data out, while accessing the hyperplanes sequentially.

The second approach recognizes that this communication does not have to take place in a sequential manner. Each processor sends and receives eight planes (either $N_\xi N_\zeta$ or $N_\eta N_\zeta$ boundary planes) of data at the outset and then performs all the computation. It would seem that the latter strategy should be more efficient than the former, but there is only a marginal improvement in our implementation.

The reason for the slight difference has to do with the way data is accessed during the triangular solves. The lower triangular solve begins at the bottom left front corner A and proceeds sequentially to the diametrically opposite top right back corner B, while the upper triangular solve follows the reverse path (see Figure 11). In the first approach, at the completion of the lower and upper triangular solves, the processors compute the right hand side sequentially in a manner akin to the lower triangular solve. In the second approach, each processor communicates eight planes of data and performs the right hand side computations as soon as it completes the triangular solves. However, even in the second approach, there is considerable idle time since the RHS computation is followed by the (sequential) lower triangular solve for the next time step. Thus the only difference between the two alternative strategies is that in one instance the processors are waiting to compute the right hand side and in the other, they are waiting to compute the lower triangular solve and the idle times are comparable.

6.4.6 IMPROVING LOAD BALANCE

It is clear that there is considerable load imbalance with the partitioning described above. This can be seen by concentrating on one processor e.g. processor P_{15} in Figure 11. This processor does not begin execution until the 'leading' edge reaches it and performs the lower triangular solve until the 'trailing' edge leaves it. At this point it begins an upper triangular solve and computes until the 'trailing' edge leaves it. It then computes the right hand side and idles till the 'leading' edge of the next triangular solve reaches it. This load imbalance places a limit on the speedup that can be achieved (see Section 6.2.2).

With a view to reduce this load imbalance, we consider two other two-dimensional partitioning strategies. Note that the ζ direction is still held in-processor. The first technique partitions the domain into diagonal strips. Load balance is enhanced by varying the width of the strips, which are narrow near the main diagonal and wider away from it. The schematic shown in Figure 13 illustrates this decomposition. The main problem with this strategy is that parallelism is reduced to one dimension and the storage requirements double in comparison with the simple 2-D partitioning described above. As Figure 13 illustrates, these strips may be stored as logically rectangular arrays, which is easily done by extending the domain as shown. This is similar to the diagonal storage scheme for sparse banded matrices. The strategy effectively reduces the size of the problem that can fit in one processor and will result in performance degradation. Using indirect addressing eliminates storage overhead, but could introduce other inefficiencies. Therefore, we do not consider this partitioning scheme any further.

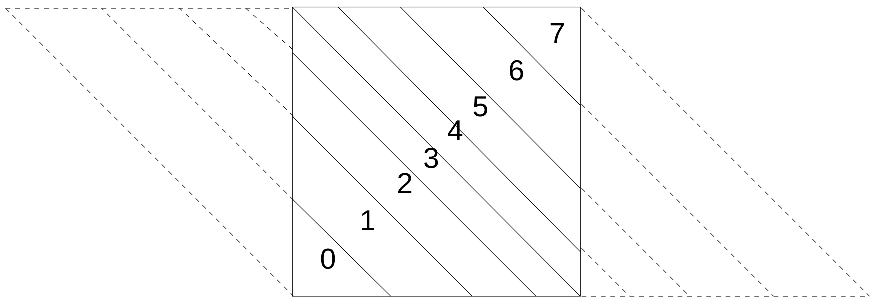

Figure 13: Stripwise diagonal partitioning with 16 processors.

The second partitioning strategy does not incur such a memory overhead and is closely related to the wrapped partitioning employed by Daoudi et al. [5] for a 2-D problem. The basic idea here is to have multiple sub-partitions assigned to each processor in a judicious manner to reduce the load imbalance. Whereas a standard 1-D wrapped partitioning is sufficient for their 2-D problem, a 2-D partitioning is needed for the three-dimensional problem on hand. A simple tensor product of two 1-D wrapped partitionings does not maintain a good load balance throughout the computation phase. The diagonal multi-partitioning idea of Naik et al.[10] shown in Figure 14 for four processors accomplishes this, but is too restrictive. It requires that all the available processors be assigned to each of the ξ and η dimensions, effectively reducing the available parallelism to one dimension.

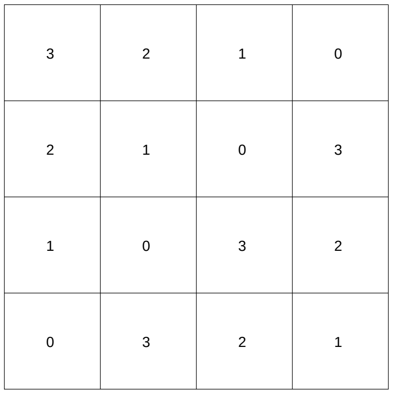

Figure 14: Diagonal multi-partitioning with 4 processors.

One solution is a one parameter multi-partitioning strategy which uses a 1-D wrapping of processors along the diagonals with the number of wraps as a parameter. Each processor is assigned a fixed number of sub-partitions equal to the number of wraps. Performance is expected to improve with increasing number of wraps until overwhelmed by communication costs. With the number of wraps set to 1, we recover the simple 2-D partitioning, and with the number of wraps set equal to the number of processors, a partitioning roughly equivalent to diagonal multi-partitioning is obtained. An example with four processors and eight wraps is shown in Figure 15. The arrows in Figure 15 indicate the direction in which the wrapping is performed.

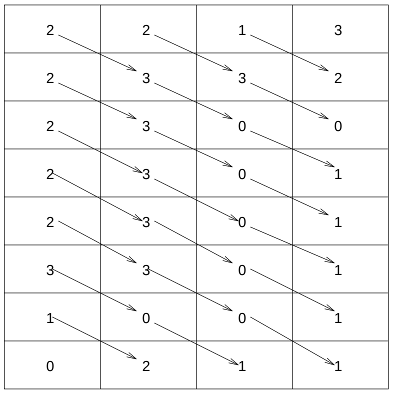

Figure 15: Wrapped multi-partitioning with 4 processors and 8 wraps.

6.4.7 MULTI-PARTITIONING ISSUES

There are several issues involved in implementing the multi-partitioning technique. Of primary concern is the communication pattern between sub-partitions. The computation phase is straightforward. All arrays are distributed equally over the sub-partitions and computations are performed by looping over the sub-partitions. Communication between the sub-partitions requires careful attention. The order in which the sub-partitions are processed is crucial to avoiding deadlocks. This issue arises whenever multi-partioning is employed.

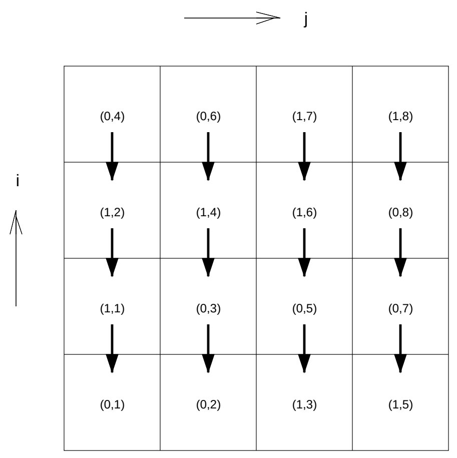

Figure 16: A simple message pattern with multi-partitioning; 2 processors and 8 wraps
.

Consider a simple southward message pattern for the multi-partitioned 2-D domain shown in Figure 16. This message pattern occurs in both the RHS and the upper triangular solve phases of the computation. The processor and sub-partition numbers are shown as tuples e.g. (0,1) means processor 0, sub-partition 1. Assume that the sub-partitions are processed sequentially within a processor. Figure 17(a) shows the communication pattern for 8 wraps using two processors. Each processor has 8 sub-partitions assigned to it. In Figure 17, the column indicates the sub-partition and the row denotes the processor. Arrows indicate the direction of communication between the sub-partitions. In Figure 17(a), notice that (1,2) is waiting to receive a message from (0,4), which cannot initiate a message until (0,3) receives a message from (1,4). (1,4) cannot initiate a message unless (1,2) and (1,3) receive their messages from (0,4) and (0,5), respectively. Hence we have a deadlock.

Deadlock occurs because sub-partitions are accessed against the flow of communication. Stated another way, dependencies between the sub-partitions create a directed acyclic forest. When sub-partitions are processed in an order that does not honor the dependencies, deadlock occurs. In this application, since sub-partitions are numbered sequentially along diagonals (Figure 15), it is possible to solve this problem by simply reversing the direction of access of the sub-partitions. The resultant communication pattern is shown in Figure 17(b) which exhibits no deadlock. Thus, for south and west messages, sub-partitions are accessed in decreasing order. Conversely, for north and east messages, sub-partitions are accessed in increasing order. Such a strategy is required even for the RHS, which has complete parallelism. A final observation is that if the sub-partitions that communicate do not differ by more than 1 in their sub-partition indices, the direction of access is not an issue, e.g. 4 processors with 4 wraps.

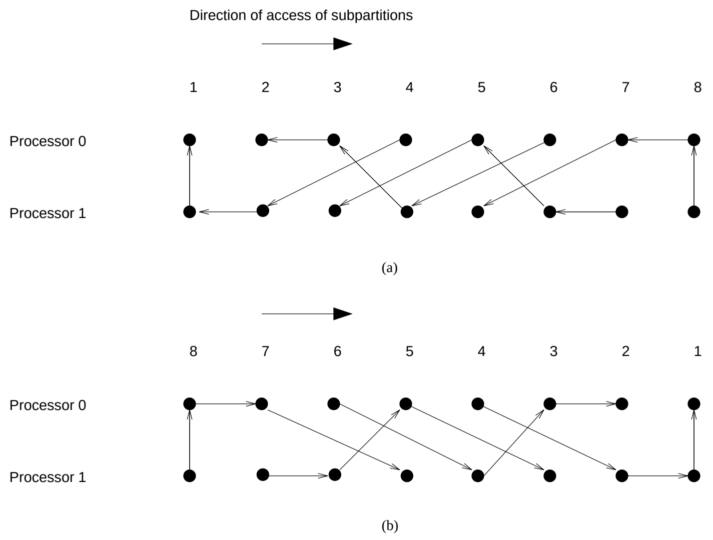

Figure 17: (a). Message pattern exhibiting a deadlock. (b). Message pattern with no deadlock.

6.4.8 iPSC/860 HYPERPLANE RESULTS

Results are expressed as the average time in seconds for a single time step of the computation using the hyperplane algorithm with the skew hyperplane mapping for various domains. There are two implementations, with and without multi-partitioning. There are upper bounds on the processors and the number of wraps since a minimum of 4 grid points is required in each of the ξ and η directions for each sub-partition.

The results from using 2-D partitioning without multi-partitioning are shown in Table 12. Comparing with Table 10, it is seen that the wavefront algorithm with a 2-D partitioning yields better performance, especially with increasing number of processors. The reason for this is given in the next section which compares the performance of the wavefront and hyperplane algorithms on the same domain.

Problem size	No. of processors							
	1	2	4	8	16	32	64	128
32^3	8.85	6.11	4.05	2.63	1.62	1.13	0.75	-
64^3	-	-	-	-	10.86	6.49	3.84	2.54
102^3	-	-	-	-	-	-	13.47	8.39
128^3	-	-	-	-	-	-	-	14.75

Table 12: Elapsed time/step (secs.) using the hyperplane algorithm on the Intel iPSC/860.

Next, the effects of multi-partitioning on performance are examined. Table 13 and Table 14 present timings obtained as the the number of processors and the number of wraps are varied for $(32 \times 32 \times 32)$ and $(64 \times 64 \times 64)$ domains, respectively. As expected, as the number of wraps are increased, the execution time decreases until communication costs begin to dominate and increases thereafter. However, the trend is not uniform. There is a bias towards square sub-partitions which consistently give better performance, e.g. 4 processors with 4 wraps, 8 processors with 2 wraps etc. The reason is that a square sub-partition ensures better load balance as the domain is swept along hyperplanes.

Finally, it should be noted that, although the results are only presented for the hyperplane algorithm, multi-partitioning should improve the performance of the wavefront algorithm as well.

Procs	No. of wraps					
	1	2	4	8	16	32
1	8.85	-	-	-	-	-
2	6.11	6.14	6.84	6.05	7.25	8.09
4	4.05	4.06	3.69	4.61	4.71	-
8	2.63	2.30	2.81	2.78	-	-
16	1.62	1.70	1.62	-	-	-
32	1.13	1.06	-	-	-	-
64	0.75	-	-	-	-	-

Table 13: Elapsed time/step in secs. using the hyperplane algorithm with multi-partitioning - $(32 \times 32 \times 32)$ domain.

Procs	No. of wraps				
	1	2	4	8	16
16	10.86	10.30	8.45	11.26	11.72
32	6.49	5.48	6.45	6.05	-
64	3.76	3.85	3.51	-	-
128	2.56	2.34	-	-	-

Table 14: Elapsed time/step in secs. using the hyperplane algorithm with multi-partitioning - $(64 \times 64 \times 64)$ domain.

Memory requirements and profile traces for the hyperplane algorithm using 64 and 128 processors on various domains are presented in Table 15. Times are presented for the RHS computation and lower and upper triangular solves. Again, over a range of domains, the two triangular solves take up nearly 85% of the time.

6.4.9 iPSC/860 ANALYSIS

The wavefront algorithm with the 2-D partitioning offers some important advantages over the hyperplane algorithm. First, it does not make use of the skew hyperplane mapping and is thus easier to implement. Second, it cuts the idle time and improves

Problem size	P	RHS		Lower		Upper		Time/step	Memory
		Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)	Perc (%)	Time (sec)	(MBytes)
32^3	64	0.100	13.3	0.322	42.9	0.322	42.9	0.750	25
64^3	64	0.561	14.6	1.649	42.9	1.625	42.3	3.840	88
	128	0.356	14.0	1.100	43.3	1.070	42.1	2.540	121
102^3	64	1.950	14.5	5.831	43.2	5.661	42.0	13.470	279
	128	1.240	14.8	3.620	43.1	3.510	41.8	8.390	361
128^3	128	2.230	15.1	6.320	42.8	6.150	41.7	14.750	582

Table 15: Profile with hyperplane algorithm.

performance without increasing the memory requirements even though the granularity is finer in the triangular solution phases. Referring to Figure 5, for a fixed k, the wavefront algorithm computes the Jacobian matrices $A_{i,j,k}$, $B_{i,j,k}$, $C_{i,j,k}$ and $D_{i,j,k}$ and stores them as 2-D arrays. Therefore each of the Jacobian matrices requires only a storage of $25 \times N_\xi \times N_\eta$ distributed over all processors. Contributions in the ζ direction are computed in parallel for the entire plane and then the wavefront algorithm with 2-D partitioning is used for the contributions in the ξ and η directions. The wavefront algorithm introduces a load imbalance when 2-D partitioning is employed which can be ameliorated by employing multi-partitioning. The important point to note is that the load imbalance does not include the time required to form the Jacobian matrices. The load imbalance is due to the sequential triangular solution phase of the SSOR simulated application. When employing the hyperplane algorithm, precomputing the Jacobian matrices is storage intensive, the Jacobian matrices need to be computed for the three-dimensional whole domain. Each Jacobian matrix requires $25 \times N_\xi \times N_\eta \times N_\zeta$ words of storage. This increase severely limits the practical problem size. However, if storage is not a consideration, the Jacobians can be precomputed in the hyperplane algorithm. Precomputing Jacobians with the hyperplane algorithm will be referred to as the pc-hyperplane algorithm.

Table 16 shows the execution times and memory requirements without multi-partitioning for the wavefront, hyperplane and pc-hyperplane algorithms for the $(64 \times 64 \times 64)$ domain. The pc-hyperplane algorithm has the best performance, but requires about 5 times as much memory as the wavefront algorithm. The wavefront algorithm thus offers the best compromise in terms of speed and memory required.

7 CONCLUSIONS

The SSOR simulated application has been implemented on three different parallel architectures, an eight processor Cray Y-MP, a 32K processor Thinking Machine Corp. CM-2 and a 128 processor Intel iPSC/860. Of the three machines, the eight processor Cray Y-MP has the best performance and the easiest implementation.

In all cases, the formation and solution of the left hand side along with the right

Method	Time/time step	Memory (MB)
wavefront	2.74	94
hyperplane	3.76	88
pc-hyperplane	2.67	400

Table 16: Comparison of wavefront, hyperplane and pc-hyperplane algorithms - ($64 \times 64 \times 64$) domain and 64 processors.

hand side calculations account for approximately 99% of the execution time within a time step. About 75% to 86% is in the formation and solution of the left hand side.

The hyperplane algorithm is implemented on all three machines. On the Y-MP, indirect addressing, using the gather-scatter hardware, is used to form a single vector from a hyperplane. On the CM-2 and iPSC/860, the skew hyperplane mapping is used to make hyperplane access more amenable. For the CM-2, the skew hyperplane mapping works well. However, on the iPSC/860, the wavefront algorithm with a 2-D partitioning is found to be better.

The wavefront algorithm with a 2-D partitioning on the iPSC/860 is able to overlap computing the Jacobian matrices of the next k plane with computing on the mesh diagonals of the current plane. This overlap can not be done on the CM-2.

A model is presented that predicts the hyperplane algorithm is better when the number of processors is greater than $\frac{12}{7}N$.

Overall, the SSOR simulated application performed better on the parallel machines than expected.

References

[1] O. Axelsson. A Generalized SSOR Method. *BIT*, 13:443–467, 1972.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon (editors). The NAS Parallel Benchmarks. Technical Report RNR-91-002, Applied Research Branch, MS T045-1, NASA Ames Research Center, Moffett Field, CA 94035, January 1991.

[3] S. Breit, W. Celmaster, W. Coney, et al. The Role of Architectural Balance in the Implementation of the NAS Parallel Benchmarks on the BBN TC2000 Computer. In *1993 ASME Fluids Engineering Conference*, Washington, June 20-24 1993. To appear.

[4] Cray Research, Inc. *UNICOS Autotasking User's Guide*, sn-2088 edition, 1989.

[5] E. Daoudi and P. Manneback. Parallel ICCG Algorithm on Distributed Memory Architecture. In *The Fifth SIAM Parallel Processing For Scientific Computing*, pages 78–83, Houston, March 25-27 1991.

[6] R. Fatoohi and S. Yoon. Multitasking the INS3D-LU Code on the Cray Y-MP. In *AIAA 10th Computational Fluid Dynamics Conference*, pages 619–626, Honolulu, June 24-27 1991.

[7] A. Greenbaum. Solving Sparse Triangular Linear Systems Using FORTRAN with Parallel Extensions on the NYU Ultracomputer Protype. Technical Report Ultracomputer Note 99, April 1986.

[8] R. Hockney and C. Jesshope. *Parallel Computers 2*. Adam Hilger, second edition, 1988.

[9] INTEL Corp. *iPSC/2 and iPSC/860 Users's Guide*, order number 311532-006 edition, June 1990.

[10] V. Naik, N. Naik, and M. Nicoules. Implicit CFD Applications on Message Passing Multiprocessor Systems. In Horst Simon, editor, *Parallel Computational Fluid Dynamics Implementations and Results*, pages 97–125. The MIT Press, 1992.

[11] Thinking Machines Co. *CM Fortran Optimization Notes: Slicewise Model*, version 1.0 edition, March 1991.